SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave

Skripta iz predmeta

Operacijski sustavi za ugrađena računala

Leonardo Jelenković

Zagreb, 2022.

Predgovor

Izgradnja ugradbenih sustava najčešće je vrlo složen proces. Sastoji se od izgradnje sklopovlja i programske potpore. Iako se u ovoj knjizi prikazuje samo izgradnja programske potpore, poznavanje sklopovlja vrlo je bitno pri izradi pojedinih elemenata programske potpore (uglavnom upravljačkih programa). S obzirom na opseg problema izgradnje ugradbenih sustava, ova knjiga pokriva samo dio tih problema, dio koji može biti dovoljan za početak. Neophodna proširenja znanja iz pojedinih područja mogu se dalje usvojiti materijalima i primjerima dostupnim na internetu.

Izgradnja jednostavnih i složenih računalnih sustava može krenuti od nule ili korištenjem već postojećih komponenata. To vrijedi i za sklopovske i programske komponente. Osim svojstava izgrađenih sustava, vrlo bitni faktor u razvoju je i vrijeme razvoja (engl. *time to market*). To se posebno odnosi na ugradbene[1] računalne sustave i sustave za rad u stvarnom vremenu, na koje se ovaj tekst prvenstveno odnosi.

Jednostavniji se sustavi ponekad mogu izgraditi i u kratkom periodu čak i da se krene od nule. Međutim, za sve ostale sustave period razvoja će se značajno smanjiti ako se krene od već djelomično izgrađenog sustava ili sustava koji se koristio za slične namjene. Takav sustav tada može poslužiti kao predložak (engl. *framework*) za izgradnju novih sustava. Različiti računalni sustavi traže razne sklopovske i programske komponente te će za njih biti potrebni različiti predlošci, s različitim komponentama i postavkama.

Jedna od mogućnosti za ostvarenje programske komponente je da se kao predložak iskoristi neki od komercijalno dostupnih sustava (engl. *commercially available off-the-shelf – COTS*), uključujući pripadnu razvojnu okolinu. Druga je mogućnost da se koriste slobodno dostupni sustavi s dostupnim izvornim kôdom, primjerice korištenjem sustava temeljenog na jezgri Linuxa te slobodno dostupnih razvojnih alata.

U prvom slučaju, kada se koristi komercijalni sustav, problem (osim eventualno cijene) može predstavljati i zatvorenost takvog sustava. Proizvođač najčešće neće dati izvorne kôdove svog sustava koje bi potencijalni inženjer (engl. *developer*) mogao prilagoditi svojim posebnim zahtjevima. U takvoj situaciji, ako je zbog drugih okolnosti ipak zaista nužno izabrati takav sustav, programer/inženjer možda ipak može svoje zahtjeve ostvariti, ali na značajno teži, složeniji i neuobičajeni način.

Kada su izvorni kôdovi dostupni, kao što je to u drugom slučaju, inženjer može teoretski potpuno prilagoditi svaku komponentu sustava svojim zahtjevima. Međutim, i u takvim sustavima može biti problema. Prvi od njih jest *složenost*. Prilagođavati jedan složeni sustav, kao što je to na primjer *jezgra operacijskog sustava*, najčešće zahtjeva iznimno dobro poznavanje takvog sustava. U protivnom, izmjena bez dubljeg poznavanja mogla bi izazvati neželjenu grešku u radu sustava.

U oba navedena primjera načina izgradnje programskog dijela računalnog sustava javlja se problem temeljitog poznavanja jezgre takvih sustava. U postupku obrazovanja, inženjer je vjerojatno dobio teoretski pregled programskih komponenata, uključujući *operacijski sustav*. S obzirom na to da je operacijski sustav vrlo složen, on se najčešće prikazuje preko modela koji dovoljno dobro opisuje samo osnovne mehanizme. Poznavanje tih mehanizam može biti dostatno za uobičajenog programera koji piše jednostavnije programe za već gotove operacijske sustave. Međutim, takvo osnovno znanje nije dostatno za inženjera koji želi u potpunosti iskoristiti mogućnosti operacijskog sustava ili koji treba prilagoditi komponente operacijskog

[1] U tekstu se koristi pojam *ugradbeni* opisujući računala pripremljena za ugradnju kao i računala koja su već ugrađena. S druge strane pojam – *ugrađeni* se odnosi samo na sustave koji su već ugrađeni. Stoga se u tekstu koristi samo pojam *ugradbeni* za sve primjene.

sustava ili koji treba prilagoditi sam operacijski sustav ili dio njegove jezgre.

Savladavanje tog znanja može biti individualno, kada inženjer sam iz dostupne literature i izvornih kôdova dobije dublji uvid u problematiku, može biti u okviru naprednijeg predmeta ili tečaja iz područja operacijskog sustava te može biti stečeno u hodu, rješavanjem tekućih problema. Prvi način, individualni rad je najteži jer dokumentacije ima napretek, što ne znači da je i potpuna. Znanje stečeno rješavanjem problema ne mora biti potpuno i opet pati od previše (nepotpune) dokumentacije. Savladavanje znanja u okviru predmeta ili tečaja koje je popraćeno prikladnim praktičnim zadacima može biti najbrži put, ako je sadržaj prikladno oblikovan.

U ovom radu predložen je jedan od pristupa podučavanja, korištenja, prilagodbe i izgradnje operacijskog sustava, tj. njegove jezgre, korištenjem primjera sustava. Osnovni ciljevi predloženog pristupa su:

- brže i temeljitije savladavanje (razumijevanje) postupaka koji se koriste u ostvarenju komponenti jezgre operacijskog sustava
- savladavanje naprednih tehnika programiranja pomoću kojih će se i lakše ostvariti potrebne operacije i koji će rezultirati znatno razumljivijim izvornim kôdovima (koji će se upravo zbog toga moći jednostavnije proširiti ili prilagoditi)
- upoznavanje s dostupnim alatima za razvoj i oblikovanje sustava (i njihove mogućnosti), uključujući i emulatore sklopovlja za koje se sustav izgrađuje
- upoznavanje s POSIX sučeljem za rad s napravama, alarmima i dretvama
- osposobljavanje za snalaženje, korištenje i rad u umjereno velikim programskim projektima (kao što je to operacijski sustav)
- savladavanje problema samostalno ili uz manju pomoć (kraća predavanja ili seminari).

Predloženi pristup zasniva se na prikazu i praktičnom radu s prikladno pripremljenim kosturima sustavima, nazvanima *Benu*[2] [Benu], sustavima koji su *inkrementalno građeni*: svaki idući sustav u nizu – *inkrement* nastaje dodatkom nove komponente ili proširenjem postojeće komponente iz prethodnog inkrementa sustava.

Početni inkrement je sustav koji samo ispisuje pozdravnu poruku *Hello World*. Idući inkrement donosi prikaz mogućeg izgleda organizacije izvornih kôdova (engl. *source tree*). Sljedeći inkrementi dodaju podršku za prekide, podršku za razne naprave, podsustave, sučelja prema korisniku itd.

Svaki inkrement se sastoji od izvornih kôdova na kojima se prikazuju pojedini mehanizmi. Posebna je pažnja posvećena odabiru redoslijeda dodavanja komponenata kako bi prijelaz s jednog inkrementa na idući bio s jedne strane što blaži, a s druge strane prikazao moguće uporabe navedenog inkrementa kao predloška za izgradnju stvarnih (ugradbenih) sustava kojima je navedena funkcionalnost dovoljna. Uz svaki su inkrement pokazani primjeri korištenja (programi).

Sustav se analizira i prikazuje u sljedećim inkrementima:

1. početni inkrement – izgradnja sustava od nule – upoznavanje s razvojnim alatima
2. upravljanje datotekama s izvornim kôdom, praćenje rada sustava
3. podsustav za prihvat prekida
4. podsustav za upravljanje vremenom

[2] *Benu* je ime ptice (simbola) u egipatskoj mitologiji. Pored mnogih značenja, ona predstavlja i stvaranje, prapočetak te je ime stoga i odabrano s obzirom na to da je ovaj sustav inkrementa samo začetak operacijskog sustava. Sustav je napravljen za edukaciju, za bolje upoznavanje detalja sustava, nije pravi operacijski sustav napravljen za korištenje. Stoga se drugo značenje imena *Benu* može izvesti kao skraćenica od *built for education, not usage*.

5. upravljanje napravama kroz unificirano sučelje
6. pokretanje operacija kroz naredbenu ljusku
7. višedretvenost
8. upravljanje spremničkim prostorom – zaštita procesa i jezgre.

Inkrementi se dodatno dijele u *faze*, s obzirom na to da se želi promjene s prethodnog inkrementa na novi uvesti postepeno. Što je razlika u inkrementima veća, veća je i podjela tog inkrementa u faze. U pravilu je zadnja faza (faza s najvećim brojem) unutar direktorija inkrementa ona faza koja predstavlja sve željene novosti tog inkrementa.

Ova skripta opisuje sam sustav Benu, uključujući korištene alate, postupke izgradnje, korišteno sklopovlje, uključujući načela rada. Opisani su sadržaji inkremenata i faza, potrebne podatkovne strukture i algoritmi. Međutim, osim opisa korištenih metoda/postupaka i slično, prikazane su i ostale mogućnosti ostvarenja, druge metode/postupci koje mogu zamijeniti ili nadopuniti postojeće.

Opis postupaka/metoda i slično je značajno razumljiviji kada se koriste primjeri. Stoga su u sam tekst ugrađeni primjeri, dijelovi izvornih kôdova, detalji korištenih sklopova (ne samo načela rada) i slično. Izgradnja ugradbenih sustava zahtijeva razumijevanje na svim razinama: od sklopovlja do operacijskih sustava.

Skripta je podijeljena na 13 poglavlja i osam dodataka. Sadržaji navedeni u dodacima su namjerno izdvojeni (nisu navedeni u poglavljima) iz nekoliko razloga. Neki dodaci ne spadaju sadržajno niti u jedno poglavlje, neki se koriste iz više, ali s različitim razinama detalja, neki će biti zanimljivi samo dijelu čitatelja i slično.

23. siječnja 2014. *Leonardo Jelenković*

Sadržaj

Predgovor **i**

1. Uvod **1**

1.1. Ugradbeni računalni sustavi 1

1.2. Operacijski sustavi 2

1.3. Ostvarenje upravljanja u ugradbenim računalima 3

1.4. Zadaci u ugradbenim sustavima 5

Pitanja za vježbu 7

2. Osnovno o pripremi programske potpore za ugradbena računala **9**

2.1. Spremnici ugradbenih računala 9

2.2. Priprema programske potpore 10

2.3. Primjeri pripreme programske potpore 11

3. Razvojna okolina **23**

3.1. Slojevita izgradnja sustava 24

3.2. Razvojni alati 25

Pitanja za vježbu 26

4. Postupak izgradnje sustava **27**

4.1. Datoteka `build.sh` 28

4.2. Datoteka `startup.S` 29

4.3. Datoteka `hello.c` 32

4.4. Primjer sata 33

Pitanja za vježbu 33

5. Organizacija kôda **35**

5.1. Datoteka `Makefile` 38

5.2. Datoteka `ldscript.ld` 51

5.3. Ispis znakova na zaslon 58

5.4. Korištenje ulazno-izlaznih naprava 59

5.5. Pomoć pri traženju grešaka 60

Pitanja za vježbu 63

6. Prekidi **65**

6.1. Prekidni sustav arhitekture x86 66

6.2. Upravljanje prekidima 68
6.3. Upravljanje prekidima sklopom *Intel 8259* 72
6.4. Ostale mogućnosti upravljanja prekidima 74
Pitanja za vježbu 76

7. Algoritmi upravljanja spremnikom 77
7.1. Statičko upravljanje spremnikom 77
7.2. Dinamičko upravljanje spremnikom 77
7.3. Sučelje jezgre za korištenje gomile 82
7.4. Liste 82
7.5. Registracija više funkcija za obradu istog prekida 84
Pitanja za vježbu 85

8. Upravljanje vremenom 87
8.1. Korištenje sklopa *Intel 8253* 88
8.2. Osnovni podsustav za upravljanje vremenom 88
8.3. POSIX sučelje za upravljanje vremenom 91
8.4. Upravljanje vremenom ostvareno u jezgri 93
8.5. Neke mogućnosti drukčijeg upravljanja vremenom 96
8.6. Nadzorni alarm 96
8.7. Upravljanje objektima jezgre 97
Pitanja za vježbu 99

9. Korištenje naprava 101
9.1. Sučelje za korištenje naprava 101
9.2. Zaslon kao naprava 104
9.3. Tipkovnica 105
9.4. Serijska veza 108
Pitanja za vježbu 109

10. Naredbena ljuska 111
10.1. Ostvarenje ugradbenih sustava do sada prikazanim postupcima 113
Pitanja za vježbu 114

11. Višedretvenost 115
11.1. Uvodna razmatranja 115
11.2. Višedretvenost ostvarena izvan jezgre 121
11.3. Višedretvenost ostvarena u jezgri 124
11.4. Sinkronizacija i komunikacija 131

11.5. Signali 137
Pitanja za vježbu 141

12. Procesi 143
12.1. Načini rada procesora x86 143
12.2. Pozivi jezgri preko programskih prekida 145
12.3. Korisnički način rada 148
12.4. Odvajanje jezgre i programa u dvije cjeline 149
12.5. Programi kao zaseban proces 150
12.6. Statički procesi 156
12.7. Dinamički procesi 156
12.8. Straničenje 156
Pitanja za vježbu 161

13. Zasnivanje ugrađenih računalnih sustava 163
13.1. Pregled nekih operacijskih sustava projektiranih za ugrađene sustave 163
13.2. Svojstva različitih tipova operacijskih sustava 169
Pitanja za vježbu 172

Dodaci 175

Dodatak A - Ostvarenje za arhitekturu ARM 177
A.1. Obilježja procesora ARM 177
A.2. Korištena arhitektura ARM 179
A.3. Posebnosti u ostvarenju sloja arhitekture 180

Dodatak B - Upute za korištenje razvojnih alata 185
B.1. Razvojni alati 185
B.2. Postavke virtualizacijskih alata 185
B.3. Git – sustav upravljanja izvornim kodovima 186

Dodatak C - Izdvojene mogućnosti C-a 191
C.1. Proširene deklaracije varijabli i funkcija 191
C.2. Makroi i druge naredbe prevoditelju 193

Dodatak D - Primjeri skripte za povezivanje 203
D.1. Sažetak o povezivanju – osnovni postupci na primjeru 203
D.2. Primjeri sustava 205

Dodatak E - Nadogradnja sinkronizacijskih mehanizama 213

E.1. Dodatne operacije sinkronizacije 213
E.2. Rekurzivno zaključavanje 215
E.3. Inverzija prioriteta 216
E.4. Ostali sinkronizacijski mehanizmi 223
E.5. Nedorečenosti u korištenju sinkronizacijskih mehanizama 226

Dodatak F - Raspoređivanje dretvi u operacijskim sustavima 227
F.1. Raspoređivanje dretvi u stvarnim sustavima 227
F.2. Raspoređivanje dretvi prema prioritetu 228
F.3. Raspoređivanje prema krajnjim trenucima završetaka 228
F.4. Podrška za raspoređivanje dretvi prema POSIX sučelju 230
F.5. Raspoređivanje nekritičnih dretvi 232

Dodatak G - Dodavanje novih raspoređivača 237
G.1. Raspoređivanje podjelom vremena 238
G.2. Raspoređivanje prema trenucima krajnjih završetaka 239

Dodatak H - Primjeri programskih zadataka za vježbu 241

Dodatak I - Zadaci za vježbu 245

Literatura 261

1. Uvod

Izgradnja ugradbenih sustava zahtijeva poznavanje svojstava takvih sustava i njihovih komponenata. U okviru ovog poglavlja opisani su osnovni pojmovi, odnosno ugradbeni računalni sustavi i njihove komponente: sklopovlje, operacijski sustav i programska potpora.

1.1. Ugradbeni računalni sustavi

Ugradbene računalne sustave[1] može se kratko definirati kao kombinaciju sklopovlja i programske podrške, uz eventualno dodatne mehaničke ili druge dijelove, načinjenu da obavlja specifičnu funkciju [Bar, 1999].

Ugradbeni računalni sustavi postaju sve značajnije područje u znanosti i industriji. Zahvaljujući napretku poluvodičke tehnologije, procesori kao osnovne komponente upravljanja ugradbenih sustava, postaju sve brži uz istodobno manju potrošnju energije i manje dimenzije. Njihova uporaba se iz tih razloga sve više proširuje na nova područja. Ugradbeni sustavi čine ogromno područje od, primjerice, najjednostavnijih igračaka pa do složenih sustava upravljanja letjelicama. Prema podacima iz 2009. godine u svijetu se godišnje proizvede deset milijardi procesora od kojih je više od 98% namijenjeno za ugradbene sustave. Zbog svoje specifičnosti ugradbeni sustavi imaju svojstva ponešto drukčija od "uobičajenih" računalnih sustava. Primjerice, ugradbeni sustavi:

- su često napajani baterijama (pa trebaju biti štedljivi);
- traže dugotrajan rad (moraju biti stabilni, pouzdani, trebaju imati ugrađen postupak oporavaka od pogreške);
- mogu upravljati i kritičnim procesima (zahtijeva se pouzdanost) – u tom slučaju se govori o *sustavima za rad u stvarnom vremenu* (kratica SRSV, engl. *real time systems – RT systems – RTS*).

Ugradbeni sustavi predstavljaju vrlo širok raspon uređaja: od jednostavnih do vrlo složenih. Programska će komponenta kod nekih biti jednostavna (upravljačka) petlja, dok će drugi složeniji sustavi zahtijevati operacijske sustave posebne namjene. Zbog njihovih svojstava, kao što su ograničene dimenzije i masovna proizvodnja, a kako bi takvi sustavi bili konkurentni pokušava se smanjiti proizvodna cijena uređaja. Smanjenje cijene može se ostvariti i jednostavnošću sklopovlja te smanjivanjem potrebnog spremničkog prostora. Zbog toga se programi koje uređaj obavlja optimiraju i obzirom na veličinu. Ako je potrebno istovremeno upravljati s više zadataka za koje se koriste različite dretve, potreban je i mehanizam koji će omogućiti potrebnu sinkronizaciju i raspoređivanje. Operacijski sustavi nude mnoštvo sinkronizacijskih i komunikacijskih funkcija. Uz to operacijski sustavi obično i upravljaju okolinom, prekidnim sustavom, spremničkim prostorom, odnosno u sebi imaju ugrađeno sučelje za upravljanje. Veličina spremničkog prostora koju zahtijeva operacijski sustav za svoje strukture podataka i programe može znatno premašiti veličinu koju zauzimaju primjenske dretve koje obavljaju zadaću uređaja te se zato često izbjegava korištenje operacijskih sustava[2].

[1] U nastavku se izostavlja riječ "računalni" jer se on podrazumijeva u ovom kontekstu, tj. koristi se skraćeni pojam "ugradbeni sustavi".

[2] Izbjegavanje korištenja operacijskog sustava u ugradbenim računalnim sustavima od strane inženjera takvih sustava je posebno izraženo, da je u šali skovana riječ *osophobia* (izvorna definicija: *osophobia n. – A common fear among embedded system programmers*).

1.2. Operacijski sustavi

Operacijski sustav je skup osnovnih programa koji se nalaze između *korisnika* računala, tj. *primjenskih programa* i *sklopovlja*. Njegova je zadaća da korisniku (primjenskim programima) pruži prikladnu okolinu za učinkovito korištenje računalnog sklopovlja.

Slika 1.1. prikazuje mjesto operacijskog sustava u računalnom sustavu. Primjenski programi jednom dijelu sklopovlja pristupaju preko operacijskog sustava, a drugome izravno (samo izvođenje instrukcija).

Slika 1.1. Slojevi računalnog sustava

Operacijski sustav je najkritičniji dio programskog okruženja nekog računalnog sustava te ga kao takvog smišljaju, izgrađuju i mijenjaju iskusni stručnjaci s ovog područja. Njegova korisnost mjerljiva je njegovim mogućnostima i brzini rada.

Različiti računalni sustavi postavljaju vlastite zahtjeve na programsko okruženje. Operacijski sustav osobnog računala, namijenjen mnoštvu aktivnosti kao što su uredski programi, pregledavanje i stvaranje multimedijalnih sadržaja te računalne igre, bitno se razlikuje od onog koje upravlja nekim proizvodnim procesom.

Zahtjevi prema programskom okruženju mogu se razmatrati i kroz povijesni razvoj računala i računarstva jer su i danas ugradbeni sustavi različitih mogućnosti, slično kao i mogućnosti računalnih sustava kroz povijest.

Prvi računalni sustavi, iako ogromnih dimenzija, bili su ograničeni mogućnostima procesora i pohrane podataka. Program, koji se na razne načine učitavao s vanjskih jedinica, imao je ugrađen dio za upravljanje okolinom. U prvotnim sustavima nije bilo mogućnosti za paralelni rad više zadataka već se na njima obavljao samo jedan program. Kad je program završio s radom i rezultati bili pohranjeni na neku vanjsku jedinicu, moglo se početi s učitavanjem i izvođenjem sljedećeg programa. Svi programi izvođeni na istom računalu imali su jedan dio programa sličan – dio za upravljanje okolinom. Struktura tih programa može se donekle opisati s dva dijela: dio koji obavlja potrebne proračune te dio koji čita podatke iz ulaznih jedinica i pohranjuje rezultate na vanjske jedinice. Drugi dio se sastoji od potrebnog broja funkcija za upravljanje okolinom te on može biti jednak za sve programe, tj. nije ga potrebno ponovno pisati za svaki od programa. Skup takvih funkcija može se smatrati jednim oblikom operacijskog sustava, mada se za taj dio sustava danas koristi pojam *upravljački programi*.

Zahtjevi za korištenjem prvih računala ubrzano su rasli što je dodatno ubrzalo razvoj računalne industrije. Računala su tada bila iznimno skupa te se u namjeri da se istim računalom koristi više osoba pristupilo osmišljavanju novog programskog okruženja. Takvo programsko okruženje treba omogućiti prividno istovremeni rad više zadataka na istome računalu. Dijeljenje računala, kako procesorskog vremena, spremničkog prostora i ulazno-izlaznih naprava, zahtijeva skup sinkronizacijskih mehanizama koji do tada nisu bili potrebni. Pored sinkronizacije javlja se potreba za mehanizmom razmjene podataka među korisnicima i među procesima.

Zaštita podataka od neovlaštenog korištenja postaje jedna od potrebnih funkcija programskog okruženja. Navedeni zahtjevi samo su neki od mnogih koji su se javljali tijekom razvoja operacijskih sustava i koji su utjecali na njihovu arhitekturu. Razvoj operacijskih sustava traje i dalje jer se i računala usavršavaju i područja njihove primjene proširuju te koncepti koji su bili optimalni za prethodnu generaciju ne moraju biti i za sljedeću. Tako se čak događa da se ponovno prihvaćaju stari koncepti (ili njihove modifikacije) koji postaju primjenjivi na novim arhitekturama.

Većina funkcija koje operacijski sustavi danas imaju potiče od prvih operacijskih sustava razvijenih na poslužiteljskim računalima. Ti su prvi sustavi postavili temelje gotovo svim sustavima koje danas koristimo, bilo na osobnom računalu u uredu ili kući ili poslužiteljima. Ipak, u području ugradbenih računala uporaba operacijskih sustava znatno je manja. Neki ugradbeni sustavi toliko su jednostavni da nemaju potrebe korištenja operacijskog sustava već je dovoljan upravljački program. Kada se operacijski sustavi i koriste u ugradbenim računalima tada su ti operacijski sustavi znatno drukčijih svojstava.

Načela rada operacijskih sustava, osnovni koncepti i slično bit će prikazani i ovdje, ali se zainteresirani čitatelji upućuju na literaturu koja dotičnu materiju podrobnije opisuju, kao što su [Budin, 2010] i [Silberschatz, 2002].

1.3. Ostvarenje upravljanja u ugradbenim računalima

Upravljanje različitim ugradbenim sustavima se međusobno znatno razlikuje. Negdje je dovoljan kratki upravljački program, dok je drugdje potreban složeni višezadaćni sustav posebnih svojstava.

Primjerice, osobno računalo bilo u uredu ili u kući, kao i radna stanica, nije osmišljeno za obavljanje samo jednog posla već mnoštvo njih. Zbog svoje opće upotrebljivosti osobno računalo ima mnoštvo primjena: uredsko računalo, radna stanica, poslužitelj, igraća naprava, sredstvo za izvođenje i stvaranje multimedijalnih sadržaja i slično. Međutim, zbog svoje opće namjene takvo računalo nema odgovarajuće mehanizme potrebne za uporabu u većini ugradbenih sustava. Kao prvo, dimenzijama i cijenom bitno odudara od zahtjeva ugradbenih sustava. Nadalje, što je i mnogo važnije, nema ugrađene mehanizme vremenske određenosti, kako ni u sklopovlju tako ni u programskoj okolini. Na primjer, kod osobnog računala gotovo je sasvim nebitno je li za prikaz nekog prozora na zaslonu potrebna sekunda, dvije, tri ili se neki proračun obavlja desetak ili više sekundi ili minuta, ali je ta vrsta neodređenosti nedozvoljena u ugradbenim sustavima. Uredski operacijski sustavi i sustavi na radnim stanicama imaju vrlo lošu podršku vremenski uređenim zadacima i odzivu na vanjske događaje pa su u većini slučajeva neupotrebljivi za ugradbene sustave (osim eventualno za one s blagim vremenskim ograničenjima).

Ugradbene sustave s obzirom na ostvarenje načina upravljanja (ostvarenje programske komponente) možemo podijeliti u nekoliko skupina:

1. upravljački program
2. neprekidiva višezadaćnost
3. višezadaćnost
4. uredski operacijski sustavi
5. operacijski sustavi posebne namjene.

Upravljački program sastoji se od jedne beskonačne petlje u kojoj se na osnovu ulaza ili varijabli stanja sustava poduzimaju određene akcije. Rad više aktivnosti ostvaruje se slijednim provjeravanjem i upravljanjem svakom od njih. Prekidi izazvani vanjskim događajima najčešće su vrlo kratki i samo mijenjaju stanje pojedinih varijabli. Drugoj inačici ove skupine pripadaju sustavi u kojima je gotovo sva aktivnost raspoređena u prekidnim procedurama, tj. obrada i upravlja-

nje se odvijaju unutar obrade prekida. Upravljački program je vrlo kratak i jednostavan te se u velikom dijelu ugradbenih sustava koristi upravo ovaj model upravljanja. Glavni nedostaci upravljačkog programa su nemogućnost pridjeljivanja prioriteta pojedinim komponentama pri upravljanju i otežano ostvarenje upravljanja nad skupom periodičnih poslova.

Kod *neprekidive višezadaćnosti* u sustavu se nalazi nekoliko *dretvi* (zadatak u izvođenju, detaljnije u poglavlju 11.), gdje svaka dretva upravlja s određenim dijelom sustava. Osim vanjskog prekida, dretvu koja se trenutno izvodi ne može prekinuti niti jedna druga dretva, već ona pri završetku trenutnog posla izravno poziva raspoređivač. Raspoređivanje i međusobna komunikacija obavljaju se izravnim pozivima dretvi te mehanizmi sinkronizacije nisu potrebni. Vanjski prekidi mogu biti ili vrlo kratki, mijenjajući samo stanja određenih varijabli ili zasebne procedure koje upravljaju pojedinim dijelovima sustava. Prednosti ovakvih sustava su u jednostavnosti izvedbe i mogućnosti pridjeljivanja prioriteta pojedinim dretvama. Osnovni nedostatak jest neprekidivost izvođenja dretve, tj. raspoređivanje se poziva isključivo na kraju rada trenutno aktivne dretve, bez obzira što se i prije toga mogu stvoriti uvjeti za nastavak rada prioritetnije dretve.

U skupinu načina upravljanja navedenim pod *višezadaćnost* podrazumijevaju se sustavi u kojima je upravljanje raspodijeljeno po dretvama različitih prioriteta. Aktivna dretva je ona s najvećim prioritetom iz skupa pripravnih dretvi. Ako se nakon prekida ili drugog događaja (primjerice funkcija sinkronizacije ili međusobne komunikacije) stvore uvjeti za nastavak rada dretve višeg prioriteta, prekida se izvođenje trenutne dretve te se prioritetnijoj dretvi omogućuje izvođenje. U višezadaćne sustave ubrajaju se svi operacijski sustavi za rad u stvarnom vremenu. U odnosu na prijašnje skupine višezadaćnost je mnogo većih mogućnosti, ali samim time i veće složenosti.

U sustavima s blagim vremenskim ograničenjima moguće je koristiti i obične, *uredske operacijske sustave*. Zbog znatne programske podrške takvi su sustavi mnogo jeftiniji za izgradnju i održavanje, ali im je upotrebljivost ograničena. Iz sličnih su razloga nastale i izvedenice takvih operacijskih sustava s puno boljom podrškom za primjenu u ugradbenim sustavima. Zbog sličnosti sustava, dodatno obrazovanje za programere takvih sustava se minimizira te se ubrzava prenošenje aplikacija na novi sustav. Takvi sustavi, međutim, zbog nastojanja zadržavanja osnovnih koncepata početnog sustava, najčešće imaju značajno dulji odziv na vanjske događaje od operacijskih sustava posebno oblikovanih za ugradbene sustave. Primjena tih sustava je zbog toga ograničena te se isti ne primjenjuju u kritičnim sustavima kontrole i upravljanja.

Operacijski sustavi posebne namjene grade se po narudžbi te se od operacijskih sustava, bilo za ugradbene ili ostale sustave, bitno razlikuju u svojim svojstvima i ponašanju. Primjeri operacijskih sustava posebne namjene su i operacijski sustavi za sustave za rad u stvarnom vremenu (primjerice [VxWorks]).

Glavni dio operacijskog sustava – njegova jezgra, može se ostvariti na nekoliko konceptualno različitih načina [Engler, 1998], [Nutt, 2000]:

- monolitna jezgra
- mikrojezgra
- proširiva jezgra
- razne hibridne kombinacije navedenih.

Kod *monolitnih jezgri* sve se funkcije operacijskog sustava izvode u sustavskom načinu rada, tj. s najvećim prioritetom te su kao takve neprekidive u svom izvođenju. Funkcije jezgre mogu se stoga međusobno usko povezati pridonoseći kompaktnosti i brzini. Monolitne jezgre su stoga najčešće vrlo brze, pogotovo u operacijskim sustavima za rad u stvarnom vremenu. Primjer monolitne jezgre jest [VxWorks] operacijski sustav, klasični primjer sustava za rad u stvarnom vremenu. Stabilnost, pouzdanost te mnoštvo funkcija sinkronizacije i međusobne komunikacije uz ostale atribute (snažno razvojno okruženje, multiplatformska podrška) čine ga vrlo popularnim izborom za uporabu u ugradbenim sustavima. Nedostaci monolitne jezgre očituju se u

teškoćama nadogradnje novim funkcijama ili promjenama nekih svojstava. Također, ako uslijed nepredviđenog tijeka izvođenja neka od jezgrenih funkcija izazove neoporavljivu pogrešku, cijeli je sustav kompromitiran, a ne samo dotična dretva ili dio sustava. Jedan od načina poboljšanja proširivosti takvih sustava jest korištenje modularnosti, tj. jezgre koja se sastoji od osnovnog dijela i dodatnih modula. Moduli se po potrebi mogu dinamički uključivati i isključivati iz jezgre. Primjer operacijskog sustava s modularnom jezgrom jest Linux [Linux].

Operacijski sustavi zasnovani na *mikrojezgri* nastoje riješiti neke probleme koji se pojavljuju kod monolitnih jezgri. Mikrojezgra se sastoji samo od skupa osnovnih funkcija, dok se ostale funkcionalnosti operacijskog sustava ostvaruju izvan jezgre, u korisničkom načinu rada. Kada se u nekoj funkciji pojavi kritična greška često je moguće zaustavljanje rada samo dotičnog procesa ili dretve, dok bi ostatak sustava mogao nastaviti normalno raditi. Dio kôda koji pripada operacijskom sustavu, a izvan je jezgre, unaprijed je definiran i nepromjenjiv, kao primjerice upravljački programi. Korisniku ili njegovu procesu nije dozvoljeno korištenje proizvoljnih programa za pristup sklopovlju već se to obavlja preko unaprijed definiranih funkcija. Tako se osigurava izvođenje samo provjerenog kôda, ali i unose ograničenja. Zbog minimalnog skupa funkcija i ostvarenja glavnine funkcija operacijskog sustava van jezgre, učestalo se mijenja način rada te su brzine rada ovih sustava lošije od sustava s monolitnom jezgrom. S druge strane prednosti su u proširivosti, održavanju i većoj stabilnosti pri ispadu pojedinih dijelova sustava. S obzirom na uporabu u ugradbenim sustavima značajna je prednost u veličini same jezgre koja može biti i za red veličine manja od uobičajenih monolitnih jezgri. Često su funkcije koje su dio mikrojezgre dovoljne za ugradbeni sustav. Primjer mikrojezgre jest Neutrino u sustavu QNX [Neutrino].

Zanimljiva rasprava o organizaciji jezgre, brzini i sigurnosti, vođena je između A. Tanenbauma i L.B. Torvaldsa [T-T debate] u kojoj se prvi zalaže za mikrojezgru dok drugi za monolitnu.

Proširiva jezgra [Engler, 1998] je jedan od oblika mikrojezgre koji prvenstveno služi za ostvarenje mehanizama zaštite u sustavu. Takva jezgra dozvoljava korisničkim programima izravan i zaštićen pristup sklopovlju. Sloj koji ona pruža vrlo je tanak u usporedbi s klasičnim operacijskim sustavima gdje se često za pristup određenom sklopovlju mora koristiti nekoliko podsustava. *Exokernel* je primjer navedene zamisli koja je ostvarena kroz nekoliko sustava visoke učinkovitosti. Takvi sustavi, međutim, nisu sustavi opće već specijalne namjene u kojoj se znatno ističu brzinom rada.

1.4. Zadaci u ugradbenim sustavima

Funkcija većine ugradbenih računalnih sustava je vrlo jednostavna i ostvaruje se kratkim programima. Međutim, zahvaljujući tehnologiji broj složenijih računalnih sustava kod kojih jednostavni programi više nisu dovoljni postaje značajan. Ostvarivanje upravljanja takvim sustavima zato zahtijeva analizu tipičnih poslova koji se u tim sustavima pojavljuju.

Sustavi za rad u stvarnom vremenu postavljaju stroga vremenska ograničenja u radu računalnih sustava koji se u njima koriste. Programi moraju pored ispravnih vrijednosti – *logičke ispravnosti*, svoje rezultate i akcije generirati u definiranim trenucima – moraju biti vremenski ispravni, zadovoljiti *vremensku ispravnost*. Zadovoljavanje vremenskih ograničenja zahtijeva detaljnu analizu problema koji se rješava, ali i metoda koje se koriste pri njegovu rješavanju, uključujući i postupke upravljanja sustavom: upravljanje sredstvima sustava, upravljanje zadacima, ...

Sustavi za rad u stvarnom vremenu (njihov računalni dio) mogu biti ugrađeni u druge (veće) sustave, tj. može ih se smatrati i ugradbenim računalnim sustavima. Isto tako, sustavi mogu biti fizički izdvojeni od sustava koje nadziru i upravljaju, a s njima vezani odgovarajućim komunikacijskim kanalima. Takvi računalni sustavi izvana mogu sličiti i na uobičajene računalne sustave – osobna računala i radne stanice. S druge strane, svi ugradbeni računalni sustavi ne

moraju imati stroga vremenska ograničenja te ih se često i ne mora svrstati i u kategoriju sustava za rad u stvarnom vremenu. Slika 1.2. prikazuje odnos sustava za rad u stvarnom vremenu i ugradbenih računalnih sustava. U jednom segmentu ugradbeni računalni sustavi ujedno spadaju i u kategoriju sustava za rad u stvarnom vremenu, ali u ostalim segmentima ne. Osim dva prikazana područja računalnih sustava, u računalne sustave spadaju i drugi sustavi (primjerice osobna računala, poslužitelji, dlanovnici, telefoni i slični koji se dijelom mogu i preklapati s navedenim na slici).

Slika 1.2. Razni računalni sustavi

Za netrivijalne sustave, svojstva zadataka su slična za sustave za rad u stvarnom vremenu i za ugradbene sustave. Zato je iduća analiza zadataka objedinjena.

Pri razmatranju ugradbenih sustava potrebno je poznavati oblik poslova koji se obavljaju – zadataka koje treba obaviti. Zadaci mogu biti periodički s unaprijed zadanim vremenima ponavljanja ili sporadični, kao primjerice reakcija na pojavu prekida. I jedni i drugu zadaci imaju neka zajednička svojstva za čije se promatranje mogu definirati određeni vremenski trenuci bitni za njihovo odvijanje. Slika 1.3. prikazuje svojstvene trenutke pri obavljanju zadatka ugradbenih sustava.

Slika 1.3. Svojstveni trenuci u životnom ciklusu jednog zadatka

Zadatak se sa svojim poslom pojavljuje u sustavu u trenutku t_d. Izvođenje može započeti u trenutku t_{mp} koje može biti i jednako trenutku dolaska. Trenutak kada zadatak počinje svoje izvođenje označeno je s t_p. Moguće je da se zadatak za vrijeme svog izvođenja prekida drugim zadatkom većeg prioriteta ili obradom prekida. U trenutku t_z zadatak završava s pridijeljenim poslom te završava s radom i nestaje iz sustava ili se privremeno zaustavlja do sljedećeg pojavljivanja posla. Do trenutka t_{kz} zadatak mora obaviti posao ili će sustav snositi određene posljedice (dogodila bi se greška). Ako je posao zadatka periodički, njegovo sljedeće aktiviranje očekuje se u trenutku $t_d + T$. Vremenska uređenost događaja sa slike mora se očuvati ako se želi stabilan rad sustava, tj. za navedene vremenske trenutke mora vrijediti uređenje: $t_d \leq t_{mp} \leq t_p < t_z \leq t_{kz}$.

Zadaća je operacijskog sustava ili nekog drugog rješenja koje se koristi umjesto njega, omogućiti održavanje navedenog vremenskog uređenja za sve zadatke u sustavu. Drugim riječima, zadaci moraju biti upravljani tako da svoje izvođenje počinju najranije u trenutku t_{mp} te da sav posao obave najkasnije do t_{kz}. Postoje razni algoritmi kako to postići, od kojih su neki vrlo jednostavni dok su drugi vrlo složeni. Odabir algoritma ovisi o uporabi. Ponegdje će i oni najjednostavniji biti sasvim dovoljni dok će drugdje biti potrebni drugi, različite složenosti. Problem upravljanja zadacima (koji u izvođenju postaju *dretve*) detaljnije je objašnjen kasnije, u pogljavlju 11..

Pitanja za vježbu 1

1. Navesti slojeve računalnog sustava.
2. Što je to *operacijski sustav*? Koja je njegova uloga u računalnom sustavu?
3. Navesti svojstva ugradbenih računalnih sustava. Usporediti ih s osobnim računalima.
4. Što su to *upravljački programi* u kontekstu ugradbenih sustava koji imaju operacijski sustav i u kontekstu onih koji nemaju?
5. Usporedite *neprekidivu višezadaćnost* i *višezadaćnost* u pogledu jednostavnosti ostvarenja te u pogledu mogućnosti koje nude.
6. Što je to *logička ispravnost*, a što *vremenska ispravnost*?
7. Što su to *sustavi za rad u stvarnom vremenu*?
8. Navesti svojstvene trenutke u životnom ciklusu zadatka.

2. Osnovno o pripremi programske potpore za ugradbena računala

Priprema programske potpore uveliko ovisi o sklopovlju ciljanog ugradbenog računala, posebice o svojstvima dostupnih spremnika. U ovom poglavlju prikazana su svojstva spremnika, općeniti postupci pripreme programske potpore za nekoliko uobičajenih primjera spremnika ugradbenih računala te primjeri pripreme ugradbenih sustava zasnovanih na jezgri Linuxa. Naglasak ovog poglavlja je postavljen na sam postupak, a ne na izradu/izmjenu izvorna kôda (što je napravljeno u idućim poglavljima).

2.1. Spremnici ugradbenih računala

Ugradbeno računalo, kao i većina ostalih tipova računala, koristi nekoliko tipova spremnika, svako sa svojim svojstvima i namjenom. Spremnike možemo po raznim kriterijima podijeliti u nekoliko kategorija. Jedna od podjela jest prema mogućnosti očuvanja podataka i nakon gašenja računala. Po toj podjeli spremnike dijelimo na trajne i privremene. Trajni spremnici očuvaju zatečene sadržaje i nakon gašenja, dok privremeni spremnici čuvaju podatke samo dok su spojeni na napajanje (dok računalo radi). Očuvanje podataka i nakon gašenja računala zahtijeva tehnologiju koja donosi i poneka ograničenja u korištenju takvog tipa spremnika. Najznačajnija su vremena pristupa podacima zapisanim u takvim spremnicima koja su osjetno veća od nego kod privremenih spremnika. Radi povećanja učinkovitosti pri radu računala se koriste privremeni spremnici, dok se podaci koji trebaju ostati i nakon gašenja računala zapisuju na trajne spremnike prije gašenja samog računala.

Trajni spremnici se dijele na one mala kapaciteta, koji se koriste samo pri pokretanju računala, i na one veća kapaciteta koji se koriste u složenijim sustavima za pohranu operacijska sustava, programa i podataka (za ostvarenje datotečna sustava s navedenim elementima). Tehnologije izrade trajnih spremnika su razne: ROM (engl. *read only memory*), EEPROM (engl. *electrically erasable programmable ROM*), flash spremnik (NAND i NOR flash) i druge. Osim navedenih, u trajne spremnike mogli bismo ubrojati i naprave za pohranu podataka kojima se ne pristupa izravno instrukcijama procesora, već preko nekih posredničkih sklopova. U ovu kategoriju spadaju tvrdi diskovi (HDD i SSD), optički diskovi, spremničke kartice i slični.

Privremeni spremnici se također izrađuju u raznim tehnologijama. Za veće brzine rada potrebni su tehnološki zahtjevniji sklopovi, koji su zbog toga i skluplji. Stoga se u jednom računalu najčešće koristi jedan veći privremeni spremnik koji se naziva radni spremnik ili glavni spremnik, a često i samo kraticom RAM (engl. *random access memory*) izrađen u običajenoj (jefitnijoj) tehnologiji te dodatni priručni spremnici (engl. *cache memory*) boljih svojstava, ali bitno manjeg kapaciteta, smještenih uz sam procesor. Priručni spremnici su pomoćni spremnici koji služe za povećavanje učinkovitosti sustava, a ne za povećanje kapaciteta spremničkog prostora. Njima najčešće upravlja sklopovlje procesora i nisu izravno dostupni preko adresa, kao radni spremnik. Ima iznimaka gdje to nije tako, gdje je upravljanje privremenim spremnikom prepušteno programima, tj. gdje je i on dostupan preko adresa.

Računala uobičajeno imaju jedan mali trajni spremnik, ROM, EEPROM ili slični (ROM u nastavku), koji sadrži program za početno pokretanje i inicijalizaciju sustava.

Kod osobnih računala se taj program naziva BIOS (engl. *basic input-output system*), odnosno UEFI (engl. *unified extensible firmware interface*) kod novijih računala. Osobna računala koriste disk za pohranu datoteka s operacijskim sustavom i ostalim programima i podacima te kod njih BIOS (ili UEFI) služi samo za početnu inicijalizaciju te pokretanje operacijskog sustava s diska.

Ugradbeni sustavi su različiti obzirom na tipove spremnika koje koriste. U svim je sustavima prisutan radni spremnik (privremeni spremnik, RAM u nastavku) koju se koristi pri radu, ali tipovi i veličine trajnih spremnika mogu biti različite.

Kod jednih se koristi samo jedan ROM u kojem je sve smješteno, od programa za početnu inicijalizaciju, operacijskog sustava do svih programa i podataka (često u komprimiranom obliku). Pri pokretanju takvih sustava jedan dio (ili i sve) iz ROM-a se kopira u RAM i tada se tek pokreće operacijski sustav koji preuzima kontrolu i pokreće potrebne programe.

Drugi, složeniji sustavi posjeduju bar dva tipa trajnog spremnika: jedan manji koji sadrži samo program za početnu inicijalizaciju sustava (ROM) te jedan veći koji sadrži sve ostalo. U takvim sustavima program za početnu inicijalizaciju po dovršetku osnovne inicjalizacije učitava i pokreće program (operacijski sustav) s većeg trajnog spremnika (diska ili sličnog spremnika većeg kapaciteta).

2.2. Priprema programske potpore

Programska potpora za ugradbeno računalo se sastoji od:

1. programa za početnu inicijalizaciju,
2. operacijskog sustava te
3. korisnih programa koji se izvode unutar operacijskog sustava.

U nastavku je prvo razmotren sustav kod kojeg se u početku sve nalazi u ROM-u.

Program za početnu inicijalizaciju, program *pokretač* ili kraće *pokretač* (engl. *boot loader*), treba pripremiti za mjesto (adresu) u ROM-u na koje će se on upisati. Pokretač se s tog mjesta i pokreće. U nekim sustavima se pokretač dijeli na dva dijela. Prvi dio samo prekopira (i po potrebi ekstrahira) sadržaj iz ROM-a u RAM te tada pokreće svoj drugi dio, ali sada iz RAM-a. Taj drugi dio tada dovršava inicijalizaciju te pokreće operacijski sustav.

Operacijski sustav treba pripremiti za mjesto s kojeg će se izvoditi, tj. za mjesto u RAM-u. Tu adresu treba prethodno proračunati uzevši u obzir veličinu operacijskog sustava, veličinu drugih podataka koji se kopiraju u RAM, početnu adresu RAM-a i slično. Ako se sustav može pripremiti da ne ovisi o početnoj adresi (korištenjem relativnih načina adresiranja), onda se može učitati bilo gdje u RAM.

S obzirom na to da operacijski sustav najčešće koristi straničenje za upravljanje spremnikom (u sustavima koji imaju sklopovsku potporu za to), pripremu programa treba prilagoditi samo operacijskom sustavu. Program će u izvođenju koristiti logičke adrese te je svejedno gdje će se učitati u radni spremnik.

U složenijim računalima kod kojih se operacijski sustav nalazi u drugim spremnicima podataka (npr. flash ili disk), pokretač treba učitati dio operacijskog sustava u RAM i tada ga pokrenuti (predati mu upravljanje).

Programska potpora za ugradbeno računalo priprema se na zasebnom razvojnom računalu (osobnom računalu ili radnoj stanici). Uglavnom je to računalo potpuno drukčije od ugradbenog za koje se programska potpora priprema. Stoga pri pripremi treba koristiti odgovarajuće alate i postupke prikladne za ciljano ugradbeno računalo (engl. *cross-platform development tools*), uzimajući u obzir instrukcijski skup podržan na tome sustavu kao i ostale posebnosti sklopovlja tog sustava.

Pri pripremi (prevođenju) programa pokretača, kao i pri pripremi operacijskog sustava razvojni alati moraju u te komponente ugraditi sve njima potrebne operacije. Ako se koriste operacije iz nekih standardnih biblioteka i te biblioteke treba uključiti u izlaznu sliku tih programa, tj. statički ih ugraditi u programe ili ih treba priložiti uz operacijski sustav kao posebne datoteke

datotečna sustava (arhive i dijeljeni elementi, `.a` i `.so` na UNIX sustavima, `.dll` na Windows zasnovanim sustavima) koje se učitavaju po potrebi – dinamički.

S druge strane, pri pripremi programa koji će se izvoditi unutar pripremljenog operacijskog sustava, program treba pripremiti za taj operacijski sustav. To znači da se mogu koristiti sučelja koje operacijski sustav pruža. Programi se na isti način (statički ili dinamički) mogu povezati s bibliotekama.

2.3. Primjeri pripreme programske potpore

U ovom poglavlju prikazan je postupak izgradnje sustava zasnovanih na Linuxu, ali bez ulaženja u detalje jezgre i ostalih korištenih sustava. Prilagodba jezgre pojedinom ugradbenom sustavu zahtijevala bi mnogo detalja i poznavanja kako sklopovlja tako i pojedinosti o samoj jezgri. Ta složenost prelazi granice ovog predmeta te su zato kao primjeri uzeti sustavi za koje već postoje odgovarajuće postavke. Ipak, za mnoštvo sustava već postoje pripremljene postavke. Treba ih samo pronaći i iskoristiti – što i nije previše složeno obzirom na dostupnost informacija i mogućnosti pretraživača.

Ovdje prikazani postupci izgradnje imaju svrhu stjecanja osnovne predodžbe o komponentama programske potpore, operacija potrebnih za njihovu pripremu, uvid u alate potrebne za izgradnju te mogućnosti pokretanja u simuliranom okruženju. Oni nisu dostatni za savladavanje problema prilagodbe programske potpore nekom ugradbenom sustavu, već samo uvod u to područje.

Nešto više informacija o samim alatima može se naći u idućim poglavljima te na internetu. U tekstu se za upravljanje alatima (programima na razvojnom računalu) pretpostavlja korištenje sustava APT (engl. *advanced packaging pool*) koji je prisutan na *Debian GNU/Linux* i sličnim sustavima, tj. koriste se naredba `sudo apt-get install ime_paketa` za dohvat i instalaciju paketa. Isto se na drugim sustavima postiže sličnim alatima.

Operacijski sustav zasnovan na Linuxu od procesora traži podršku za straničenje. Za sustave s vrlo jednostavnim procesorima koji nemaju podršku za straničenje pripremljena je pojednostavljena inačica Linuxa bez tog zahtjeva – *uClinux* [uClinux]. Za pripremu sustava temeljenog na njemu koriste se slični postupci kao i za pripremu sustava temeljenih na Linuxu te nisu zasebno opisivani u nastavku.

2.3.1. Priprema jezgre operacijska sustava Linux

Izvorni kôdovi jezgre operacijska sustava Linux (najnovija inačica, ali i sve prethodne) nalaze se na webu. U trenutku pisanja ovog teksta, zadnja inačica jezgre nosi oznaku: 4.20.12. Izvorni kôdovi se mogu dohvatiti u arhivama nekoliko različita formata (.tar.gz, .tar.bz2, .tar.xz). Arhiva s kodom je velika oko 100 MB, a kada se raspakira oko 900 MB.

Priprema direktorija za rad, dohvat arhive i njeno raspakiravanje može se napraviti sa sljedećih nekoliko naredbi.

```
$ pwd
/home/user
$ mkdir -p elinux/izv-kod
$ cd elinux/izv-kod
$ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.12.tar.xz
$ tar -vxJf linux-6.1.12.tar.xz
$ cd ..
$ ln -s izv-kod/linux-6.1.12 linux
```

Prethodnim naredbama kôd je dohvaćen sa zadane adrese i raspakiran u direktoriju `elinux/izv-kod/linux-6.1.12`. Potom je u početnom direktoriju `elinux` napravljena veza na taj

direktorij imena `linux` (radi lakšeg dohvata i kraćeg zapisa naredbi idućih koraka).

S obzirom na to da se program `tar` učestalo koristi za rad s arhivama s izvornim kôdom poželjno je poznavati njegove najosnovnije naredbe. Prethodna naredba uključuje zastavice `vxJf`. Zastavica `v` nalaže da se pri radu ispišu i dodatne informacije (engl. *verbose*). Zastavica `x` označava naredbu raspakiravanja (engl. *extract*). Pakiranje, tj. sažimanje zadalo bi se zastavicom `c` (engl. *create*). Zastavica `J` označava arhivu tipa `xz` (`j` označava arhivu tipa `bzip2`, a `z` arhivu tipa `bz`). Zadnja zastavica `f` kaže da se koristi datoteka koja slijedi kao izvor arhive (`tar` može arhivu čitati i preko standardna ulaza, tj. cjevovoda). Primjerice, želi li se direktorij `linux` pakirati u arhivu `linux.tar.bz2` treba zadati naredbu: `tar -cjf linux.tar.bz2 linux/`.

Prevođenje izvornih kôdova jezgre (u sliku s jezgrom koju se može pokrenuti) se obavlja posebnim skupom alata. Alati ovise o ciljanoj arhitekturi za koju se jezgra priprema. U nastavku će naprije biti prikazana priprema jezgre za arhitekturu x86, a potom za jedan sustav zasnovan na procesoru ARM.

Prije samog prevođenja izvornog kôda treba odabrati postavke jezgre. Odabir postavki je tema za sebe (prikazana na drugim mjestima) i neće se analizirati ovdje. Naglašene su samo one postavke koje su različite od pretpostavljenih, koje je potrebno uključiti radi određenih ciljeva.

Priprema jezgre za arhitekturu x86

Priprema jezgre za arhitekturu x86 može biti napravljena s pretpostavljenim postavkama (bez dodatnih podešavanja). Takve postavke se uzimaju naredbom `make defconfig`. Postavke se mogu promijeniti na nekoliko načina. Jedan od njih jest preko izbornika. Sljedeći niz naredbi prikazuje navedene operacije.

```
$ cd linux
$ make distclean
[...] (poruke)
$ make defconfig
[...] (poruke)
$ make menuconfig
[...] (prozor u kojem se mijenjaju postavke)
```

Ako neka od navedenih naredbi javi grešku, najvjerojatnije je to zbog nedostataka alata ili biblioteka. U tom slučaju pogledati dodatak B. kao i napomene u nastavku o tome kako te dodatke postaviti.

Naredba `make defconfig` uzima u obzir razvojni sustav na kojem se jezgra priprema i nastoji stvoriti jezgru sličnih svojstava kao one koja se trenutno koristi na razvojnom sustavu. Primjerice, ako je to 64-bitovni sustav postavke će se preuzeti iz datoteke `x86_64_defconfig`.

Ako bi se željelo napraviti 32-bitovnu inačicu jezgre (za x86) na 64-bitovnom sustavu potrebno je napraviti neke pripreme koje već spadaju u kategoriju prevođenja za drugi sustav. Jedno od rješenja jest najprije definirati varijablu okoline `ARCH` i u nju postaviti i386 (`export ARCH=i386`) pa tek onda raditi prema gornjim uputama. Drugo uključuje odabir datoteke sa željenim postavkama: `make i386_defconfig`. Međutim, za takvo prevođenje potrebne su i odgovarajuće 32-bitovne biblioteke. One se za `gcc` nalaze u paketu `gcc-multilib`.

Ako naredba `make menuconfig` (koja nije potrebna ako promjene postavki nisu potrebne ili rade se na druge načine) treba dohvatiti i dodatni paket `libncurses-dev`.

Postupak prevođenja pokreće se s `make`. Kada razvojni sustav ima više procesora, prevođenje se može ubrzati traženjem da se datoteke prevode paralelno korištenjem zastavice `-j N`, gdje `N` predstavlja broj paralelnih prevođenja datoteka (npr. `make -j 3`).

Po završetku prevođenja (koje može potrajati nekoliko minuta ili više) jezgra je spremna, a gdje se nalazi ispiše se na kraju prevođenja (prikazan je primjer za 64-bitovnu jezgru nastalu

na takvom sustavu).

```
$ make
[...]
  BUILD   arch/x86/boot/bzImage
Kernel: arch/x86/boot/bzImage is ready  (#1)
$
```

Za korišteni sustav i postavke, slika izgrađena sustava (jezgre) nalazi se u datoteci `arch/x86/boot/bzImage`. Veličina same jezgre (kad se raspakira u RAM) je nešto više od 8 MB.

Za sada nam je slika jezgre dovoljna. Međutim, mnogi upravljački programi su pripremljeni (prevedeni) kao moduli, ali nisu uključeni u navedenoj slici. Ako bismo ih željeli uključiti u naš sustav onda ih moramo ili uključiti u sliku jezgre (ne prevesti ih kao module već uključiti u jezgru – onemogućiti podršku za module u postavkama) ili staviti u datotečni sustav pa da se mogu učitati po potrebi (inicijalizaciji naprava).

Samo jezgra nije dovoljna za bilo kakav sustav. Za obavljanje bilo kakvog posla potrebni su programi koji će koristiti jezgru radi obavljanja potrebnih operacija. Takve programe treba pripremiti za tu jezgru i dodati uz nju, najčešće na datotečnom sustavu koji takav sustav koristi. Ipak, radi prikaza da prevedena jezgra nešto radi u nastavku se koristi uobičajeni "Hello World" program, kao jedini program koji jezgra pokreće nakon svoje inicijalizacije.

Datoteka: hello-world.c

```
1 #include <stdio.h>
2 void main()
3 {
4         printf ( "Hello World!\n" );
5         while(1);
6 }
```

Program na svom kraju ima beskonačnu petlju iz razloga što će taj program biti početni program za jezgru te kao takav ne smije nikad završiti.

Priprema programa za pokretanje i njegovo pakiranje u jednostavni datotečni sustav može se obaviti naredbama:

```
$ cd ..
$ gcc -static hello-world.c -o hello-world
$ echo hello-world | cpio -o --format=newc > initramfs
```

Prva naredba prevodi sam program[1]. Zastavicu `-static` potrebno je postaviti tako da se u sam program ugrade sve potrebne biblioteke koje program koristi jer dinamičko povezivanje neće biti moguće – nema datotečna sustava s bibliotekama koje bi se mogle naknadno (dinamički) učitati.

Druga naredba stvara jednostavnu arhivu od izlazne datoteke `hello-world`. Ta se arhiva može koristiti kao jednostavni datotečni sustav koji jezgra Linuxa prepoznaje.

Stvorena jezgra i jednostavni program mogu se pokrenuti korištenjem emulatora QEMU. Jedan od načina na koji se to može napraviti jest naredbom[2]:

```
$ qemu-system-x86_64 -machine accel=tcg -kernel linux/arch/x86/boot/bzImage -initrd
 initramfs -serial stdio -display none -append "rdinit=/hello-world console=ttyS0"
```

Zastavicom `-kernel` zadaje se datoteka s jezgrom operacijskog sustava (osnovni dio programa koji se pokreće u simuliranom okruženju). Zastavicom `-initrd` postavlja se dodatak (datoteka) koji QEMU također učita u RAM (i pritom po potrebi i raspakira). Dijelom `-serial stdio` nalaže se povezivanje serijskog pristupa simuliranog računala s konzolom u kojoj se

[1] Koristiti zastavicu `-m32` ako se stvara 32-bitovni sustav na 64-bitovnom razvojnom računalu.
[2] Koristiti program `qemu-system-i386` ako se izgradio 32-bitovni sustav (jezgra i program).

QEMU pokreće – podaci koje simulirano računalo šalje na serijski pristup prikazuju se konzoli, a tekst koji se piše u konzoli prosljeđuje na serijski ulaz simulirana računala. Zadnji dio naredbe koji započinje sa zastavicom `-append` nalaže QEMU da dio u navodnicima proslijedi simuliranom sustavu (jezgri Linuxa) kao parametre naredbenog retka. Prvi parametar, `rdinit=/hello-world` definira početni program koji jezgra treba pokrenuti. Drugi parametar definira da umjesto zaslon (konzolu simuliranog sustava) treba koristiti serijski pristup (tako se ispis prosljeđuje na konzolu u kojoj je sam QEMU pokrenut).

Među porukama koje se ispišu na konzoli, a koje su uglavnom poruke pojedinih dijelova jezgre, treba se naći i poruka `Hello World` koju ispiše program koji jezgra pokreće, negdje pri kraju, kao u prikazanom primjeru u nastavku.

```
[...]
[    3.701239] Run /hello-world as init process
Hello World!
```

Priprema jezgre za ARM

Radi mogućnosti jednostavna pokretanja u emulatoru QEMU odabran je sustav temeljen na ARM Versatile Express ploči. [vexpress-a9]

Prije samog prevođenja jezgre potrebno je obrisati sve već prevedene dijelove jezgre kao i postavke.

```
$ pwd
/home/user/elinux
$ cd linux
$ make distclean
```

Za razvoj Linuxa za procesore ARM korištenjem razvojnog računala zasnovanog na procesoru x86 (ili drugom) potreban je dodatan skup alata. Za razvoj jezgre Linuxa dovoljna je posebno pripremljena inačica alata *gcc* u paketu `gcc-arm-linux-gnueabi`. Kao što i samo ime sugerira, radi se o skupu alata za pripremu jezgre i programa za procesore ARM i Linux korištenjem EABI (engl. *embedded application binary interface*) sučelja za komunikaciju među programa i jezgre. EABI definira načine prijenosa parametara u funkcije, poziv jezgrine funkcije i slične operacije.

Pri prevođenju jezgre Linuxa za drugi sustav (engl. *cross-compile*) potrebno je to naglasiti u svakom trenutku. To se može napraviti ili dodavanjem potrebnih postavki preko varijabla okoline ili pak njihovim navođenjem u svakoj naredbi prevođenja. Primjerice, ako varijable `ARCH=arm` i `CROSS_COMPILE=arm-linux-gnueabi-` nisu postavljene u varijablama okoline (s `export` u ljusci `bash`) u svakoj naredbi ih je potrebno dodati (npr. `make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-`). U ovom prikazu je odabran pristup sa zadavanjem tih vrijednosti preko varijabli okoline.

```
$ export ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
```

Početne postavke za sustav za koji se priprema jezgra definirane su u `vexpress_defconfig` datoteci koju se postavlja kao početne postavke prevođenja naredbom:

```
$ make vexpress_defconfig
```

S obzirom na to da EABI nije uključen u početnim postavkama definiranim u gornjoj datoteci, to treba napraviti na neki od načina. U nastavku je to napravljeno preko izbornika.

```
$ make menuconfig
```

U postavkama se ARM EABI omogućava preko izbora:

1. *Kernel Features*

2. uključiti *[*] Use ARM EABI to compile the kernel*
3. uključiti *[*] Allow old ABI binaries to run with this kernel*

Samo prevođenje pokreće se naredbom `make`.

```
$ make
  [...]
  Kernel: arch/arm/boot/zImage is ready
  [...]
$
```

Za pripremljenu jezgru pripremit će se isti program kao i kod x86 inačice. Isti program `hello-world.c` sada se prevodi posebnim alatom:

```
$ cd ..
$ arm-linux-gnueabi-gcc -static hello-world.c -o hello-world
```

Za razliku od prevođenja jezgre, gdje su sve potrebne biblioteke za jezgru zadane u samom izvornom kôdu jezgre, pri prevođenju programa za sustav koji je pogonjen jezgrom Linuxa taj program ili treba sam ostvariti sve što treba ili treba dohvatiti takve biblioteke. U primjeru programa koristi se funkcija `printf` koja je ostvarena u biblioteci `libc`, ali za sustav ARM-Linux. Ako navedena biblioteka već nije dohvaćena u sklopu paketa *gcc-arm-linux-gnueabi* treba ju naknadno skinuti (npr. u paketu `libc6-dev-armel-cross`). Isti problem bi bio i za x86 da na razvojnom sustavu već nisu bile prisutne te biblioteke (koje su sastavni dio razvojnog okruženja).

Slika te datoteke prikladne za učitavanje s jezgrom stvara se na isti način kao i prije.

```
$ echo hello-world | cpio -o --format=newc > initramfs
```

Pokretanje u emulatoru zahtijeva i poseban program (na računalu s procesorom x86 se pokreće program koji emulira sustav ARM):

```
$ qemu-system-arm -M vexpress-a9 -kernel linux/arch/arm/boot/zImage -dtb linux/arch
/arm/boot/dts/vexpress-v2p-ca9.dtb -initrd initramfs -serial stdio -display none -
append "rdinit=/hello-world console=ttyAMA0"
```

Među porukama koje se ispišu na konzoli, a koje su uglavnom poruke pojedinih dijelova jezgre, trebala be se naći i poruka `Hello World` (možda treba malo pričekati, obzirom da nema upravljačkih programa i da može zapeti negdje neko vrijeme).

```
[...]
Run /hello-world as init process
Hello World!
```

Emulacija sustava ARM unutar QEMU se može prekinuti s [Ctrl]+[A] te [X] (iako će za ovu priliku i [Ctrl]+[C] biti dovoljno).

2.3.2. Priprema osnovnog skupa alata – BusyBox

Program koji ispisuje samo jednu poruku `Hello World` nije baš koristan program. Za različite primjene trebaju različiti programi. U ovome je poglavlju pokazano kako pripremiti minimalni skup uobičajenih alata dostupnih na UNIX sustavu. Jedan skup takvih alata pripremljen u izvornom kôdu, a koji će se koristiti za demonstraciju, jest BusyBox [BusyBox]. Prikazat će se priprema samo za sustav ARM jer je vrlo slično i za x86 (čak i jednostavnije).

Naredbe za dohvat i početna podešavanja su u nastavku.

```
$ pwd
/home/user/elinux
$ cd izv-kod
```

```
$ wget https://busybox.net/downloads/busybox-1.36.0.tar.bz2
$ tar xjvf busybox-1.36.0.tar.bz2
$ cd ..
$ ln -s izv-kod/busybox-1.36.0 busybox
$ cd busybox
$ make distclean
$ export ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
$ make defconfig
$ make menuconfig
```

U osnovi je prevođenje skupa alata BusyBox identično prevođenju programa `hello-world.c`. Razlika jest u tome što se ovdje izvorni kôd sastoji od puno datoteka te koristi sličan postupak za prevođenje kao i jezgra Linuxa. Ali u suštini je prevođenje jednako – program se priprema za Linux na sustavu ARM – samo je još dodatno potrebno definirati procesor, ali ne i ostatak sustava kojem program ionako pristupa preko jezgre.

Prije prevođenja potrebno je promijeniti nekoliko postavki. S obzirom na to da u ovom primjeru nismo pripremili dijeljene biblioteke, u postavkama je potrebno postaviti da se program statički povezuje sa svim potrebnim operacijama:

1. *BusyBox Settings*
2. *Build Options*
3. uključiti *Build BusyBox as a static binary (no shared libs)*

Prevođenje se pokreće s `make install`. Po prevođenju datotečni sustav će biti pripremljen u direktoriju `_install`. Direktoriji koji su tamo ipak ne čine potpun datotečni sustav. Pri pokretanju takva sustava mnoge operacije neće raditi. Neke od njih se mogu popraviti dodavanjem direktorija i datoteka prema sljedećim uputama.

```
$ make install
$ cd _install
$ mkdir -p proc sys dev etc/init.d
$ cat <<EOF > etc/init.d/rcS
#!/bin/sh
mount -t proc none /proc
mount -t sysfs none /sys
/sbin/mdev -s
EOF
$ chmod +x etc/init.d/rcS
```

Navedeno se sada priprema kao datotečni sustav.

```
$ find . | cpio -o --format=newc > ../rootfs.img
$ cd ..
$ gzip rootfs.img
```

Korištenjem prethodno izgrađene jezgre za istu arhitekturu ARM navedeni se program može pokrenuti kao i prije (umjesto programa `hello-world`).

```
$ cd ..
$ pwd
/home/user/elinux
$ qemu-system-arm -M vexpress-a9 -kernel linux/arch/arm/boot/zImage -dtb linux/arch
/arm/boot/dts/vexpress-v2p-ca9.dtb -initrd busybox/rootfs.img.gz -serial stdio -
display none -append "root=/dev/ram rdinit=/sbin/init console=ttyAMA0"
```

Nakon pokretanja (i možda čekanja da se sustav dovede u radno stanje) dolazi se do konzole s naredbama busybox-a.

```
Please press Enter to activate this console.
/ # ls
bin      etc      proc     sbin     usr
```

```
dev      linuxrc  root     sys
/ # help
Built-in commands:
------------------
    . : [ [[ alias bg break cd chdir command continue echo eval exec
    exit export false fg getopts hash help history jobs kill let
    local printf pwd read readonly return set shift source test times
    trap true type ulimit umask unalias unset wait
/ #
```

Mnogo toga nedostaje, ali minimalni skup naredbi je ipak tu.

Potpuniji datotečni sustavi se izgrađuju nešto složenije. Među elementima koji nedostaju sustavima izgrađenim prethodnim postupcima treba ponovno spomenuti upravljačke programe za naprave koji su za jezgru pripremljeni kao moduli, ali nisu ugrađeni u samu sliku jezgre. Ideja u korištenju mehanizma modula jest u tome da se oni koji nisu neophodni samo pripreme i pohrane na datotečni sustav te da se učitaju pri radu jezgre tek kada postanu potrebni. Drugi bitan element su biblioteke. I sam je BusyBox statički povezan s obzirom na to da se pri pokretanju na pripremljenom sustavu ne nalaze takve biblioteke. Kada bi one bile pripremljene ni njega ne bi trebalo statički povezivati.

Skup operacija koje BusyBox pruža je poveći, ali ipak ograničen. Za veći skup trebalo bi pripremiti i potrebne programe. Jedan od alata koji može poslužiti za izradu datotečna sustava jest i *Buildroot* [Buildroot]. On uključuje i druge elemente, može pripremiti i samo razvojno okruženje, prevesti jezgru i ostale programe. Međutim, može ga se koristiti i samo radi izgradnje datotečna sustava. Obzirom na veći broj postavki koji je potrebno postavljati, njegovo korišenje nije ovdje opisano.

Umjesto pripreme sustava korištenjem izvornih kôdova, prema gornjim ili sličnim postupcima, vrlo često se na internetu može naći gotov sustav sa svim potrebnim alatima za ciljani ugradbeni sustav. Po potrebi se na takav sustav još mogu dodati i dodatno potrebni programi.

2.3.3. Priprema programa za pokretanje sustava uz U-Boot

U prethodnim primjerima izgrađeni sustav je pokretan u simuliranom okruženju korištenjem programa QEMU. QEMU je programe (jezgru i sliku datotečna sustava) učitavao iz datoteka.

U stvarnim ugradbenim sustavima slika sustava se nalazi u ROM-u (ili sličnom obliku trajna spremnika). Radi pokretanja takva sustava potreban je dodatni program pokretač koji će sliku sustava raspakirati iz ROM-a u RAM te potom pokrenuti. Načelna organizacija spremnika takva sustava prikazana je slikom 2.1.

Slika 2.1. ROM i RAM ugradbena sustava

Početne adrese ROM-a i RAM-a (adrese A_1 i A_2) moraju biti poznate prilikom pripreme sustava. Najčešće ROM započinje na adresi 0 tako da se pokretanjem računala signalom RESET započinje s pokretanjem programa pokretača na početku ROM-a. U nekim sustavima se te adrese mogu promijeniti postavkama pojedinih upravljačkih registara. U takvim je sustavim uobičajeno da se nakon početne inicijalizacije i kopiranja potrebnih dijelova sustava u RAM radi

promjena adresa tako da RAM počinje na adresi 0, a ROM se premješta na neku veću adresu.

Jedan od popularnih programa pokretača za primjenu u ugradbenim sustavima jest U-Boot [U-Boot]. On trenutno ima podršku za mnoštvo ugradbenih sustava, ali se i dalje razvija i ugrađuje podrška za nove sustave. Kao i prethodno korišteni elementi programske potpore i on je slobodno dostupan u obliku izvorna kôda.

Priprema slike za učitavanje u ROM sastoji se od programa pokretača, slike jezgre i slike datotečna sustava. Ta tri elementa treba prikladno zapakirati i učitati u ROM.

Priprema jezgre i slike datotečna sustava je u osnovnim koracima pokazana u prethodnim odjeljcima. U nastavku najprije slijedi prikaz koraka za pripremu U-Boota.

```
$ pwd
/home/user/elinux
$ cd izv-kod
$ wget https://source.denx.de/u-boot/u-boot/-/archive/v2023.01/u-boot-v2023.01.tar.
bz2
$ tar xvfj u-boot-v2023.01.tar.bz2
$ cd ..
$ ln -s izv-kod/u-boot-v2023.01 u-boot
$ cd u-boot
$ export ARCH=arm CROSS_COMPILE=arm-linux-gnueabi-
$ make distclean
$ make vexpress_ca9x4_defconfig
$ make all
```

Slika U-Boota (koji je ovdje pripremljen da se može pokretati u QEMU-u) nalazi se u datoteci `u-boot`. Provjera ispravnosti može se ispitati pokretanjem u emulatoru QEMU.

```
$ qemu-system-arm -M vexpress-a9 -nographic -kernel u-boot
U-Boot 2023.01 (Feb 15 2023 - 21:03:36 +0100)

DRAM:  128 MiB
WARNING: Caches not enabled
Core:  18 devices, 10 uclasses, devicetree: embed
Flash: 64 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
=>
```

Za interaktivan rad s U-Bootom treba se s ugradbenim sustavom spojiti preko serijske veze. Tada se u njemu mogu zadavati naredbe. U gornjem primjeru, moguće je odgovarajućim naredbama ispitati stanje sustava, sadržaje spremnika i slično. Također je moguće preuzeti sliku sustava preko mreže jednim od podržanih protokola (TFTP, NFS, RARP/TFTP, BOOTP/TFTP).

U-Boot je pomoćni program koji se uglavnom koristi kod ugradbenih sustava. Smisao njegova korištenja je u učitavanju operacijska sustava i datotečna sustava iz ROM-a u RAM i njihovo pokretanje (u sustavu koji nema drugi medij za spremanje podataka kao što je disk). Slika jezgre i datotečna sustava pripremljena je u prethodnim poglavljima (datoteke `linux/arch/arm/boot/zImage` i `busybox/rootfs.img.gz`). Sada je potrebno te datoteke (uz U-Boot) posložiti u jednu sliku koja će se učitati u ROM, a pri pokretanju će ju U-Boot kopirati u RAM.

Načelna priprema slike za ugradbeni sustav

Pretpostavimo radi jednostavnosti da imamo ugradbeni sustav s ROM-om od 16 MB i s većim RAM-om. Neka se pri pokretanju sustava ROM nalazi na adresi 0, a RAM na adresi

0x01000000, od 16. MB (zanemarimo za sada što to ne vrijedi za korišteni sustav vexpress_ca9x4). U-Boot će pri pokretanju prekopirati obje slike u RAM (na adrese koje zadamo) i tek onda pokrenuti Linux.

U prethodnim poglavljima dobivene su slike: U-Boot i Linux jezgra, svaka slika veća od 5 MB, slika datotečna sustava nešto iznad 1 MB. Linux jezgru i sliku datotečna sustava treba prvo pripremiti u formatu u kojem ih U-Boot može identificirati i koristiti. To se može napraviti korištenjem alata `mkimage` iz skupa alata `u-boot-tools`. Navedeni alat će nešto malo povećati veličine tih slika, ali neznatno. Međutim da bi ga mogli pokrenuti moramo znati za koje adrese pripremamo te slike. Slike bi mogli staviti jednu odmah do druge (u ROM-u, u RAM-u), ali obzirom da su prve dvije veće od 5 MB u nastavku je korišteno poravnanje na 6 MB. Tako odabrane adrese adrese U-Boot-a, jezgre i datotečna sustava prikazane su tablicom 2.1.

Tablica 2.1. Veličine pojedinih elemenata i njihov smještaj u spremnicima

datoteka	veličina slike	adresa u ROM-u	adresa u RAM-u
`u-boot`	5603620	0x00000000	–
`zImage`	5243776	0x00600000	0x01000000
`rootfs.img.gz`	1139613	0x00C00000	0x01600000

Sad se mogu slike jezgre i datotečna sustava pripremiti za te adrese sljedećim naredbama.

```
$ mkimage -A arm -C none -O linux -T kernel -d linux/arch/arm/boot/zImage \
-a 0x01000000 -e 0x01000000 zImage.uimg
Image Name:
Created:      Wed Feb 15 21:07:32 2023
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    5243776 Bytes = 5120.88 KiB = 5.00 MiB
Load Address: 01000000
Entry Point:  01000000
$
$ mkimage -A arm -C none -O linux -T ramdisk -d busybox/rootfs.img.gz \
-a 0x01600000 -e 0x01600000 rootfs.uimg
Image Name:
Created:      Wed Feb 15 21:28:09 2023
Image Type:   ARM Linux RAMDisk Image (uncompressed)
Data Size:    1139613 Bytes = 1112.90 KiB = 1.09 MiB
Load Address: 01600000
Entry Point:  01600000
$
```

Kada bismo u ROM stavljali te datoteke jednu iza druge, a ne kako je prikazano tablicom 2.1., onda bismo sliku cijela sustava dobili naredbom:

```
$ cat u-boot.bin zImage.uimg rootfs.uimg > flash.bin
```

Ovo je jedna od mogućnosti, ali onda adrese potrebne pri pokretanju nisu okrugle. To zapravo i nije problem, ali da se prikaže izgradnja slike s proizvoljno postavljenim dijelovima koristi se postupak u nastavku uz adrese iz tablice 2.1.

```
$ dd if=/dev/zero of=flash.bin bs=1 count=16M # 16 MB
[...]
$ dd if=u-boot/u-boot of=flash.bin conv=notrunc bs=1 seek=0 # od početka
[...]
$ dd if=zImage.uimg of=flash.bin conv=notrunc bs=1 seek=6M # od 6. MB
[...]
$ dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=12M # od 12. MB
[...]
```

Ovako pripremljenu sliku se onda posebnim alatima učita u ROM ugradbenog sustava. Nedostaju upute U-Boot-u kako da pokrene dotični sustav. Ukoliko sustav ima podršku za serijsku vezu, mogli bi se na njega spojiti preko nje i "ručno pokrenuti" sustav.

```
=> setenv bootargs 'root=/dev/ram mem=128M rdinit=/sbin/init console=ttyAMA0'
=> bootm 0x00600000 0x01000000
```

Ili bi trebali u konfiguraciju U-Boot-a postaviti zadane adrese i argumente za Linux (npr. u datoteku `include/configs/vexpress_ca9x4.h`).

Npr. za razmatrani sustav (kada bi memorije bile prema pretpostavkama), treba prilagoditi datoteku `u-boot/include/configs/vexpress_ca9x4.h`, dodati `CONFIG_BOOTARGS` i `CONFIG_BOOTCOMMAND`.

Isječak kôda 2.1. Promjene u u-boot/include/configs/vexpress_ca9x4.h

```
1 #define CONFIG_BOOTARGS          \
2         "root=/dev/ram mem=128M rdinit=/sbin/init console=ttyAMA0"
3 #define CONFIG_BOOTCOMMAND       "bootm 0x00600000 0x01000000"
```

Da bi navedeno imalo učinka potrebno je ponovno izgraditi U-Boot te ga ubaciti u sliku sustava `flash.bin`.

Za pokretanje u stvarnom sustavu trebalo bi ispitati koje su adrese ROM-a i RAM-a te prilagoditi gornje instrukcije.

Za pokretanje u simulatoru preko QEMU-a bilo bi također potrebno poznavati adrese koje QEMU koristi u odabranom modelu sustava, tj. gdje učitava sliku u memoriju. Potom bi te adrese trebalo koristiti pri pripremi slika.

2.3.4. Priprema programske potpore za novi sustav

Potpuno novi sustav, različit od postojećih sustava za koje već postoje pripremljeni elementi programske potpore vrlo su rijetki. Novi su sustavi većinom slični nekim postojećim sustavima. Stoga priprema programske potpore za takve sustave može započeti pronalaskom tih sličnih sustava za koje već postoji programska potpora.

Prilagodba programske potpore će morati obuhvatiti isključivanje elemenata koji nisu prisutni na novom sustavu, prilagodbom postojećih elemenata parametrima i ponašanju nova sustava te dodavanjem (izgradnjom kôda) elemenata koji nedostaju.

Ovakve operacije zahtijevaju temeljito poznavanje novog sustava, ali i svih elemenata programske potpore koje se priprema. Drugim riječima, ovakav poduhvat je vrlo zahtjevan. Radi ubrzanja ovakva razvoja preporuča se razmatranje već postojećih sustava kao primjera te na taj način naučiti kako izgraditi nedostajuće elemente. Slična ideja je potakla stvaranje i o ovih materijala i primjera sustava koji su za njega pripremljeni i prikazani u idućim poglavljima.

2.3.5. Okruženja i razvojne okoline za ugradbene računalne sustave

Osim navedene izgradnje "od nule", mogu se koristiti i razvojna okruženja koja nude nešto više. Danas postoji dosta takvih projekata (i sve ih je više). U nastavku su kratko opisni OpenWRT, projekt Yocto, Ubuntu Core i Windows 10 IoT Core. Za sustave s puno većim ograničenjima na procesnu snagu i spremnik koriste se posebno pripremljeni operacijski sustavi (više o tome u okviru 13. poglavlja).

OpenWRT (`https://openwrt.org/`) je razvojno okruženje uglavnom usmjereno na mrežne uređaje, ali mogu se i drugi ugradbeni sustavi temeljiti na njemu. Temeljen je na jezgri Linuxa i ima razvijenu podršku za mnoštvo mrežnih uređaja.

Yocto Project (`https://www.yoctoproject.org/`) je razvojno okruženje za izgradnju pro-

gramske komponente ugradbenih računala. Linux je i u ovom projektu temelj. Naglasak ovog okruženja jest na prenosivosti i izgradnji u slojevima. Primjerice isto programsko okruženje koje je pripremljeno za neku sklopovsku arhitekturu se jednostavno pripremi za drugu, izmjenom slojeva koji se tiču same arhitekture, dok su ostali slojevi nepromijenjeni.

Ubuntu Core kao i Windows 10 IoT Core su posebno pripremljene inačice operacijskih sustava i razvojnih okruženja usmjerene prema ugradbenim računalima, manjih zahtjeva prema sklopovlju nego stolne inačice navedenih operacijskih sustava.

3. Razvojna okolina

Izgradnja operacijskih sustava je vrlo složen proces. I u današnje vrijeme najčešće korištene operacijske sustave u računalnim sustavima koji nas okružuju skoro pa bismo mogli pobrojati na prste (*Windows*; Linux*; OS X*; Android; iOS;* ...). Razloga tome ima nekoliko, od cijene izrade novog operacijskog sustava, tržišnom natjecanju, potrebnim normama i sličnim tehničkim, ekonomskim, socijalnim i političkim utjecajima.

Operacijski sustav je vrlo složen. Stoga ga ne izgrađuje jedna osoba nego poveći tim. U slučaju besplatnih operacijskih sustava sa slobodno dostupnim kodom (sustavi temeljeni na jezgri Linuxa i slični, primjerice GNU projekt), "tim" koji na njemu radi je poveći. Osim operacijskog sustava koji služi kao jezgra, programska potpora sastoji se od mnoštva korisnih programa. U slučaju besplatno dostupnih programa s dostupnim izvornim kodom, broj osoba uključen u takve projekte je vrlo velik. Kako tako velika razvojna ekipa, koja je raspodijeljena po cijelom svijetu surađuje na izradi programske potpore? Iako većina njih volontira u tom razvojnom procesu, ipak moraju postojati osobe koje su "odgovorne" za pojedini dio programa. Takve osobe odlučuju (najčešće na osnovu prihvata promjena preko raznih oblika komunikacije) koje će se promjene uključiti u iduću inačicu programa.

Programeri uključeni u razvoj zajedničkih projekata trebali bi se pridržavati nekih pravila. Neka su pravila posebno vezana uz pojedini projekt (npr. stil pisanja koda), a neka su pravila općenita. Jedno od važnih općih pravila jest što manje uvišestručavati kod: ako potrebna operacija već postoji u okviru neke biblioteke (koja je nastala zbog potrebe drugih projekata) onda iskoristiti tu operaciju preko te biblioteke. Prednosti ovakva pristupa nisu samo u smanjenju obima posla unutar trenutnog projekta (nije potrebno ostvariti navedenu operaciju) već i veća razina sigurnosti u ispravnost rada te operacije. Naime, ako se ta biblioteka koristi i u drugim sustavima, moguće greške će se prije uočiti te i ispraviti – ažurirat će se i sama biblioteka. Međutim, potrebno je pratiti razvoj takve biblioteke, jer se ponekad dogodi da neke operacije promjene sučelje ili čak ako više nisu potrebne drugim projektima se miču iz projekata (idućih inačica biblioteka). Opću povezanost raznih programa (alata, biblioteka, ...) može se vidjeti prilikom instalacije novih programa, kada se osim potrebnih moraju instalirati i neki drugi koje potrebni koriste. Slično je i pri izgradnji programa iz izvorna koda – oni često trebaju popriličan broj specijaliziranih alata i biblioteka, ali koji su besplatno dostupni. Slične pristupe treba koristiti i pri razvoju i samo jednog projekta: treba uočiti koje se operacije ponavljaju i ostvariti ih na jednom mjestu, a ne uvišestručavati kod.

Sustav Benu nije izgrađen da bude konkurencija nekom od postojećih sustava. To bi bilo iznad mogućnosti autora. Ipak, postupna izgradnja jednog sustava s većinom osnovnih elemenata koje pravi operacijski sustavi imaju može poslužiti kao podloga razumijevanju mehanizama koji se koriste u pravim sustavima, pogotovo onim namijenjenim za ugrađene sustave. Taj postupak može ukazati i na (ne)prikladnost nekih mehanizama. Na primjer, mogu se ocijeniti neki postupci za korištenje u vremenski kritičnim sustavima, gdje nam je više od učinkovitosti i brzine bitna vremenska određenost te tako između nekoliko različitih mehanizama odabrati one sa zadovoljavajućim svojstvima.

Benu – izgrađeni sustav prikazan u ovoj skripti, ima samo osnovne funkcionalnosti i time je bliže operacijskim sustavima za ugrađena računala koja imaju znatno manje mogućnosti (funkcionalnosti) od ostalih operacijskih sustava. Osim toga, sustav je izgrađivan uzimajući u obzir jednostavne izvedbe procesora, odnosno koristi se samo najosnovnije sklopovlje (iako je pisan za arhitekturu s više mogućnosti). Benu je "igračka" koja služi za savladavanje osnovnog znanja te poticanje interesa za rad sa stvarnim sustavima.

3.1. Slojevita izgradnja sustava

Sustav je početno izgrađen za arhitekturu Intel i386 (skraćeno na i386 i x86 u tekstu). Kasnije je proširen i na arhitekture zasnovane na procesorima ARM. Pri prikazu sklopovlja kao i pri ostvarenju pojedinih operacija koristi se arhitektura i386. Arhitektura ARM i ostvarenje istih operacija prikazano je u Dodatku A. Iako arhitektura Intel i386 nije tipičan predstavnik za ugrađene sustave (ali i to se mijenja!), njena je prednost u obilju razvojnih alata i emulatora, a može se koristiti i izravno (bez emulatora) na gotovo svakom *osobnom računalu*. S obzirom na to da je kôd pisan u slojevima, za korištenje na drugim arhitekturama dovoljno je prilagoditi, tj. proširiti samo jedan sloj — *sloj arhitekture* (engl. *hardware abstraction layer – HAL*), kao što je to i prikazano na primjeru arhitekture ARM.

Slika 3.1. prikazuje strukturni prikaz izgrađenog sustava s nazivima slojeva prikazanim u zagradama. Nazivi slojeva odgovaraju direktorijima u strukturi izvornog kôda.

Slika 3.1. Slojevita izgradnja – slojevi i direktoriji

Specifičnosti sklopovlja, kao što su to upravljanje prekidima te ispisom na zaslon i ostalim ulazno izlaznim napravama, korištenjem prikladnih *upravljačkih programa* (engl. *device driver*), sakrivene su u *sloju arhitekture* (ostvarenom u direktoriju *arch* te prema njemu i imenovanome). Sloj *arch* svoje usluge nudi i jezgri i aplikacijama preko sučelja definiranog u `include/arch`.

Sloj jezgre (engl. *kernel*) ostvaruje osnovne elemente sustava: upravljanje ulazno-izlaznim napravama (na višoj razini), upravljanje spremnikom, procesorom, vremenom (sustavom alarma), dretvama, procesima i dr. Sučelje koje jezgra nudi višim slojevima definirano je u `include/kernel`.

Sloj sustavskih funkcija, tj. sloj *api*[1], ostvaruje potrebne operacije korištenjem sučelja *jezgre*. Često korištene funkcije jezgre su ovdje pojednostavljene kroz dodatno sučelje za programe. Također, ovdje su ostvarene složenije operacije za koje je potrebno više jezgrenih funkcija. Sučelje koje *api* nudi višim slojevima definirano je u `include/api`.

Na najvišoj razini nalaze se *aplikacije*, tj. *programi* (sloj *programs*), koji ostvaruju željene operacije sustava (prema korisniku ili drugim sustavima). Programi ne koriste izravno jezgru, već koriste sučelje sloja sustavskih funkcija (*api*).

Operacije koje su nezavisne i koje se koriste u više slojeva, kao štu su operacije nad nizovima znakova, operacije nad brojevima i složenijim strukturama podataka (na primjer, liste), odvojene su u jednu *biblioteku* (*lib*).

Sučelja su većinom jednosmjerna $arch \rightarrow kernel \rightarrow api \rightarrow programs$, ali ima i iznimaka. Na primjer, ako se iz sloja *arch* želi nešto ispisati (poruka o grešci) ipak je to potrebno napraviti kroz sloj jezgre (koja ostvaruje funkciju `kprintf`).

Sustav koji se prikazuje (Benu) je nastao korištenjem ideja i dijelova kôda iz raznih izvora.

[1] *API* je skraćenica od engleskog termina: *application programming interface*.

Sustav nije građen da bude optimalan, već je samo jedan od mogućih načina izgradnje sustava. Odluke o izborima u oblikovanju sustava vođene su načelima jednostavnosti izvedbe, odvajanje slojeva (prema slici 3.1.), primjena jednostavnijih algoritama, minimiziranje korištenja specifičnosti sklopovlja, modularnost, korištenje svih mehanizama programskih jezika radi čitljivosti, smanjivanja složenosti i izbjegavanja dupliciranja kôda, prilagođenost za uporabu u ugrađenim sustavima i sustavima za rad u stvarnom vremenu.

U stvarnim sustavima jezgra ima puno više koda koji je dodatno podijeljen u direktorije. Primjeri takve organizacije mogu se pronaći u [Linux], [BusyBox], [U-Boot], [uClinux].

3.2. Razvojni alati

Kao programski jezik odabran je C, uz vrlo malo ipak neophodnog *asemblera* – strojnog koda zapisanog u mnemoničkom obliku.

Alati korišteni za izradu sustava (za arhitekturu Intel i386) su:

- *gcc* – prevoditelj (engl. *compiler*) iz paketa *GCC (GNU Compiler Collection)*
- *ld* – povezivač (engl. *linker*), *GNU linker* iz paketa *GNU Binutils*
- *GNU Make* – alat za pokretanje prevođenja izvornih kôdova i izgradnju sustava
- QEMU – virtualizacijski alat za pokretanje izgrađenog sustava.

Razvojno okruženje pokretano je iz Linuxa jer su svi potrebni alati na njemu već uobičajeno prisutni ili ih se može vrlo jednostavno dodati. Konkretno, za izradu Benua je korišten Ubuntu sustav u virtualnom okruženju, korištenjem VMware Player programa. Međutim, isto bi tako trebao raditi i na ostalim sustavima (distribucijama) Linuxa.

Slika izgrađenog sustava (kao slika jezgre u ELF formatu) korištena je kao medij za pokretanje sustava uporabom programa QEMU unutar Linuxa.

Umjesto navedenih, mogu koristiti i drugi virtualizacijski alati (primjerice *VirtualBox*) ili sustavi izravno pokretani Linuxom (bez emulacije). Bitno je jedino da imaju tražene alate. Alati se mogu i naknadno dodati prikladnim naredbama (primjerice program `prog` sa `sudo apt-get install prog` u sustavima koji koriste APT (engl. *advanced packaging tool*) (na Ubuntu) ili sa `yum install prog` pomoću alata *YUM*(engl. *yellowdog updater, modified*) (na Fedori). Svi korišteni alati mogu se besplatno dobaviti sa stranica proizvođača (uglavnom GNU).

Odabrani alati definiraju neke svoje norme u pojedinim segmentima izgradnje sustava – izvornim kôdovima u C-u i asembleru, način povezivanja objektnih datoteka. S obzirom na to da je i za prevođenje asemblera odabran *gcc* koristi se njegova konvencija u pisanju kôda (AT&T konvencija, uz posebne oznake). U slučaju odabira nekog drugog alata (primjerice *NASM*) bilo bi potrebno promijeniti dio asemblerskog kôda.

Izvorni kôdovi za pojedine inkremente izgradnje (`ChapterNN` u strukturi izvornih kôdova) operacijskog sustava dostupni su na Webu [Benu].

Od dodatnih alata, jedan od vrlo bitnih je okruženje za pisanje i mijenjanje izvornih kôdova. I među besplatnim programima ima ih mnogo različitih sposobnosti. Neki od njih su: *Gedit*, *Atom*, *Eclipse*, *KDevelop*, *vim*, *Geany*, *Visual Studio Code* i slični. Jedan od vrlo dobrih alata za usporedbu izvornih kôdova (primjerice za praćenje izmjena između različitih inkrementa projekta) je *Meld* (slika 3.2.).

Prilikom izgradnje kôda nastoji se očuvati čitljivost kôda. Zato se (gotovo) uvijek poštuje granica od najviše 80 znakova u retku. Sintaksa prati strukturu kôda korištenjem tabulatora (uobičajene veličine – ekvivalent 8 razmaka). Kôd je dosta "prozračan", ubacuju se razmaci između elemenata naredbi kao i prazni redci da bismo razdvojili elemente datoteka. Za označavanje komentara u kôdu koristi "stariji" način označavanja s `/* */`. Komentiranje funkcija nije pot-

Slika 3.2. Prikaz usporedbe direktorija korištenjem programa *Meld*

puno, ali je rađeno (uglavnom) sukladno uobičajenim normama tako da bi se u budućnosti, uz male prilagodbe, mogli koristiti alati za automatsko generiranje dokumentacije.

Imena funkcija i varijabli donekle prate sloj i podsustav u kojem se koriste. Na primjer, interne funkcije jezgre započinju s `k_`, sučelje jezgre prema programima sa `sys__` te sučelje sloja *arch* s `arch_`. Nastojalo se sakriti (učiniti nedostupnim) sve funkcije i varijable koji nisu potrebne izvan pojedinih dijelova (datoteka). Međutim, i dalje postoji poveći broj funkcija i varijabli koje su dostupne svugdje. To bi donekle moglo ograničavati izradu programa, jer se imena tih varijabli i funkcija ne mogu koristiti za nove varijable i funkcije (npr. ni u običnom C programu ne smije se koristiti varijabla naziva `printf` jer je to jedna od već definiranih funkcija).

Pitanja za vježbu 3

1. Navesti slojeve sustava Benu.
2. Navesti potrebne alate za izgradnju programske komponente. Koji se programi (alati) koriste pri izgradnji i pokretanju sustava nastalih na osnovu Benua?
3. Navesti prednosti i nedostatke korištenja virtualizacijskih alata za pokretanje:
   a) razvojnog računala
   b) izgrađenog sustava (na osnovi inkremenata iz sustava Benu).

4. Postupak izgradnje sustava

Početni inkrement[1] izgradnje sustava služi za upoznavanje s kôdom, načinom prevođenja i pokretanjem stvorenog sustava. Prvi je inkrement zato vrlo jednostavan i u konačnici proizvodi samo sustav koji ispisuje poruku "Hello World".

Razmatranje upravljanja sklopovljem (na niskoj razini) u ovom prikazu je dosta ograničeno jer ovdje nije cilj baviti se detaljima određenog sklopovlja već prikazati uobičajene postupke upravljanja. Također, ne razmatraju se svi načini rada procesora (kao osnovne komponente računalnih sustava), već samo uobičajeni način rada u kojem je dostupna većina mogućnosti procesora.

Pri uključivanju računala kontrolu najprije preuzimaju programi spremljeni u trajni spremnik – (EP)ROM (engl. *erasable programmable read-only memory*). Ti se programi označavaju kraticom BIOS (engl. *Basic Input-Output System*). Zadaća tih programa je dvojaka: prvo trebaju ispitati ispravnost komponenata sustava (spremnik, tvrdi disk, tipkovnica, ...), a potom omogućiti pokretanje operacijskog sustava, nudeći operacije dohvata podataka s raznih medija.

Pokretanje operacijskog sustava se najčešće obavlja u dva koraka. U prvom se koraku pod upravljanjem BIOS-a prvo učitava te potom pokreće program (engl. *boot loader*) koji će tada učitati osnovne dijelove jezgre s nekog medija u glavni spremnik. Nakon učitavanja kontrola se predaje jezgri, koja potom dovršava učitavanje, provjerava dostupnost sklopovlja i slično te konačno postavlja sustav u operativno stanje.

Pokretanje izgrađenog sustava može se u emuliranom okruženju napraviti na nekoliko načina. U Benu su prikazana dva načina:

1. korištenjem emulatora QEMU i njegove mogućnosti učitavanja slike sustava iz ELF datoteke te
2. korištenjem programa GRUB (punim imenom: *GNU grand unified bootloader*) za učitavanje sustava kada se sustav priprema za CD.

Korištenje GRUB-a opisano je u direktoriju `util`. Pritom se pripremljena slika sustava u formatu ELF smješta na CD (njegovu sliku) zajedno s ostalim potrebnim datotekama za GRUB. Pri pokretanju, GRUB se prvi pokreće (BIOS ga učitava). Korištenjem datoteke s postavkama GRUB-u se nalaže da učita i pokrene operacijski sustav koji se izgrađuje, a čija se slika također smješta na CD. U trenutku preuzimanja kontrole, procesor se nalazi u zaštićenom načinu rada (engl. *protected mode*). Ovaj je način izrade slike potreban kada se želi sustav pokrenuti na stvarnom sklopovlju (snimanjem CD-a) ili u emulatoru koji ne podržava izravno učitavanje slike iz formata ELF.

Datoteke za prvi inkrement za arhitekturu i386 su:

- `startup.S`
- `hello.c`
- `build.sh`

Prve dvije (`startup.S` i `hello.c`) sadržavaju izvorni kôd, dok `build.sh` sadrži upute ljusci kako izgraditi sustav.

Izgradnja sustava sastoji se od nekoliko koraka. Za izgradnju Benu koriste se sljedeći koraci:

1. prevođenje izvornih kodova u objektne datoteke (`.o` datoteke)

[1] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_01_Startup/01_Startup/i386`.

2. stvaranje slike sustava (`.elf` datoteke) prikladnim spajanjem objektnih datoteka.

Samo prevođenje izvornih kodova u objektne datoteke također se izvodi u nekoliko koraka. Sa stanovišta programera ova se operacija može podijeliti na dva dijela (dva koraka). U prvom se koraku (engl. *preprocessing*) priprema kod za pravo prevođenje: uključuju zaglavlja navedena na početku datoteke, izračunavaju svi makroi na mjestima poziva (uvrštavaju njihove vrijednosti na mjesta poziva) i slično. U drugom se koraku pokreće pravo prevođenje koda pripremljenog u prvom koraku (sa svojim podkoracima, kao što su leksička, sintaksna i semantička analiza te generiranje koda i optimiranje). Svaka datoteka s izvornim kodom (`.c` ili `.asm`) se zasebno prevodi (navedeni koraci se ponavljaju za svaku takvu datoteku).

4.1. Datoteka `build.sh`

Prevođenje se pokreće zadanom skriptom `build.sh` (primjerice iz naredbene linije s `./build.sh`). Ako se kao argument doda i `qemu` onda će po izgradnji sustav odmah i pokrenuti u emulatoru QEMU. Skripta se sastoji od definicija varijabli, provjeri parametara naredbene linije te pokretanja programa (*gcc*, *ld*, ...).

U nastavku je opisana datoteka `build.sh`.

Isječak kôda 4.1. Chapter_01_Startup/01_Startup/i386/build.sh

```
1  #!/bin/sh
2
3  # usage: ./build.sh [qemu|clean]
4  #
5
6  PROJECT=hello.elf
```

Varijabli okoline `PROJECT` (koja se stvara ako ne postoji) pridružuje se vrijednost `hello`. Varijabla se koristi u ostatku datoteke.

Isječak kôda 4.2. Chapter_01_Startup/01_Startup/i386/build.sh

```
8  #Compile if required
9  if [ $# -eq 0 ] || ( [ $1 = "qemu" ] && [ ! -e $PROJECT ] ); then
10
11 CFLAGS="-m32 -march=i386 -Wall -Werror -ffreestanding -nostdlib -fno-stack-
   protector -fno-pie"
12 LDFLAGS="-melf_i386 -e arch_start -Ttext=0x100000"
13
14 #compile
15 gcc -c startup.S $CFLAGS
16 gcc -c hello.c $CFLAGS
17
18 #link
19 ld startup.o hello.o -o $PROJECT $LDFLAGS
```

Zadane izvorne datoteke se prevode korištenjem programa *gcc*. Povezivanje se obavlja programom *ld*. Zastavice koje se daju programima *gcc* i *ld* opisane su u tablici 4.1.

U ostatku skripte će se zaustaviti izvođenje skripte u slučaju greške (pri izgradnji slike sustava) ili pokrenuti slika u emuliranom okruženju (ako je skripta pokrenuta s parametrom `qemu`) ili obrisati sve izgrađene datoteke (ako je skripta pokrenuta s parametrom `cleanall`).

Pokretanjem s `./build.sh qemu` pokreće se program QEMU u kojem se emulira izgrađeni sustav, a koji u ovom koraku ispisuje samo pozdravnu poruku `Hello World!`.

Tablica 4.1. Zastavice za *gcc* i *ld*

zastavica	značenje
-c	prevođenje u objektni oblik bez naknadnog povezivanja (engl. *compile*, zastavica je zadana izravno uz gcc, nije u CFLAGS)
-O3 (*gcc*)	optimizacija razine tri (najveća moguća)
-m32	izlazni kôd stvoriti za 32-bitni procesor
-Wall	prikazati sva upozorenja (pri prevođenju)
-ffreestanding	pretpostaviti zasebno prevođenje, gdje uobičajena pravila ne moraju vrijediti, kao što je, na primjer, nepostojanje uobičajenih biblioteka, nema početne funkcije programa – main
-nostdlib	ne koristiti uobičajene biblioteke
-fno-stack-protector	ne dodavati kôd za zaštitu stoga
-O3 (*ld*)	optimizacija razine tri (u trenutnom ostvarenju *ld* ignorira brojku)
-melf_i386	izlazni kôd stvoriti za 32-bitni procesor i386 u ELF formatu [ELF]
-e arch_start	početna (*entry*) funkcija/adresa je arch_start
-Ttext=0x100000	povezati tako da se pretpostavi učitavanje kôda na adresu 0x100000 pri pokretanju (tamo će ga QEMU/GRUB i učitati)

Isječak kôda 4.3. Chapter_01_Startup/01_Startup/i386/build.sh

```
21 #if an error occured => exit
22 if [ ! $? -eq 0 ] ; then
23         exit
24 fi
25
26 echo Created System image: $PROJECT
27
28 fi #"compile"
29
30 if [ $# -gt 0 ] && [ $1 = "qemu" ]; then
31         echo Starting...
32         qemu-system-i386 -m 2 -machine accel=tcg -kernel $PROJECT
33
34 elif [ $# -gt 0 ] && [ $1 = "cleanall" ]; then
35         echo Cleaning...
36         rm -rf *.o $PROJECT
37 fi
```

4.2. Datoteka startup.S

Datoteka startup.S sadrži informacije i instrukcije potrebne za preuzimanje kontrole nad sustavom. Pretpostavlja se da će program koji učitava sliku sustava koristiti *multiboot* specifikaciju [Multiboot]. Ta specifikacija nalaže postavljanje određenih vrijednosti u sliku sustava (zaglavlje), ali i podatke koji će biti na raspolaganju sustavu nakon pokretanja, tj. preuzimanja kontrole od programa za učitavanje.

Iako su i izvorni kôdovi komentirani, dijelovi izvornih kôdova bitni za razumijevanje i ovdje su

navedeni i dodatno opisani.

Isječak kôda 4.4. Chapter_01_Startup/01_Startup/i386/startup.S

```
3  /*! Multiboot constants (basic) */
4  #define MULTIBOOT_HEADER_MAGIC  0x1BADB002
5  #define MULTIBOOT_HEADER_FLAGS  0
```

Prethodne konstante potrebne su prema *multiboot* specifikaciji.

Isječak kôda 4.5. Chapter_01_Startup/01_Startup/i386/startup.S

```
7  /* stack, startup function */
8  .extern print_hello
```

Oznakom `.extern` se navode globalne varijable koje su izvorno definirane u drugim datotekama, a koje se koriste u ovoj. Varijabla `print_hello` jest ime funkcije koja će se pozvati nakon početne inicijalizacije, a koja je definirana u datoteci `hello.c`.

Isječak kôda 4.6. Chapter_01_Startup/01_Startup/i386/startup.S

```
10 /* this code must be first in image for boot loader to find it easy */
11 .section .text
12
13 /* entry point (required for boot loader) */
14 .global arch_start
15
16 /* 32 bit alignment is required for following constants */
17 .align  4
```

Pri prevođenju, različiti se elementi izvorne datoteke smještaju u razne *odjeljke* (engl. *section*). Naredba `.section` definira odjeljak u koji se pri prevođenju (*gcc* pokrenut sa zastavicom `-c`) smještaju naredbe i podaci koje ju slijede (do druge naredbe `section`). Oznaka `.text` se uobičajeno koristi za odjeljke s instrukcijama te gornja linija definira da sve što slijedi iza treba staviti u odjeljak `.text`.

Naredba `.global` označava simbol `arch_start` kao globalni, vidljivi i izvan ove datoteke. Ovaj je simbol potreban pri povezivanju (navodi se u zastavicama povezivača). Istim načinom bi se mogla označiti i neka druga adresa (varijabla/labela) i omogućiti korištenje izvan ove datoteke (primjerice pozivanje funkcije ostvarene u asembleru iz C kôda).

Naredba `.align 4` definira poravnanje na 4 okteta (32 bita) koje je potrebno radi poravnanja idućeg zaglavlja, a koje se traži pri učitavanju (prema *multiboot* specifikaciji).

Isječak kôda 4.7. Chapter_01_Startup/01_Startup/i386/startup.S

```
19 /* Multiboot header */
20 multiboot_header:
21         /* magic */
22         .long   MULTIBOOT_HEADER_MAGIC
23         /* flags */
24         .long   MULTIBOOT_HEADER_FLAGS
25         /* checksum */
26         .long   -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
```

Naredba `.long` kaže da se na tom mjestu nalazi podatak tipa `long` (32 bita) čija je vrijednost zadana u nastavku naredbe. Prethodni kôd tako na tri uzastopne spremničke lokacije pohranjuje tri 32-bitna broja. Navedeno zaglavlje, prema *multiboot* specifikaciji mora sadržavati *magični broj* te zastavice koje od programa za učitavanje slike traže određenu strukturu podataka o samom sustavu u kojem se sve pokreće. S obzirom na to da su zastavice u ovom koraku jednake nuli, nisu postavljeni dodatni zahtjevi.

Isječak kôda 4.8. Chapter_01_Startup/01_Startup/i386/startup.S

```
28 /* THE starting point */
29 arch_start:
30         /* stack pointer initialization */
31         mov     $stack, %esp
32
33         /* starting status register - EFLAGS register */
34         pushl   $0
35         popf
36
37         call    print_hello
38
39         /* stop: disable interrupts and suspend processor */
40         cli
41 loop:   hlt
42         jmp     loop
```

Oznaka (labela) `arch_start` definira početak kôda kojim se preuzima kontrola nad sustavom. Oznaka `arch_start` se pretvara u adresu iduće instrukcije (`arch_start` je na neki način varijabla kojoj se automatski pridjeljuje vrijednost adrese iduće instrukcije).

Po preuzimanju kontrole nad sustavom, najprije je potrebno inicijalizirati kazaljku stoga (registar `esp`) jer se on intenzivno koristi i pri pozivima potprograma i pri prekidima. Također, potrebno je postaviti odgovarajuće zastavice u registar stanja procesora (sve nule su za sada odgovarajuće). Varijabla `stack`, definirana na kraju datoteke, predstavlja početak prostora zauzetog za stog. S obzirom na to da u arhitekturi x86 kazaljka stoga pokazuje na vrh stoga (zadnji postavljeni podatak na stogu) te da stog raste prema nižim adresama, u kazaljku stoga je potrebno postaviti adresu za jedan (zapravo 4) veću od zadnje spremničke lokacije zauzete za stog. Na taj će način prvo stavljanje na stog početi puniti stog od najvećih adresa.

Instrukcija `mov` će kopirati prvi operand (vrijednost adrese `stack` definirane na kraju datoteke) u drugi operand (registar `esp`). Budući da sufiks nije zadan koristit će se pretpostavljeni tip podataka `long` (32 bita), kao da je postavljen sufiks `l` na instrukciju `mov` u `movl`.

Ovdje se primjećuje korištenje AT&T pravila koje primjenjuje *gcc*, a gdje se kao odredište koristi zadnji argument instrukcije (za razliku od Intelova pravila gdje je odredište prvi argument).

Iduće dvije instrukcije (linije 34 i 35) postavljaju nulu u registar stanja na način da najprije postave nulu na stog (`pushl $0`), a potom instrukcijom `popf` sa stoga uzmu vrijednost i prepišu registar stanja. Registru stanja se ne može izravno pristupiti, već jedino "neizravno" korištenjem posebnih instrukcija.

Instrukcija `call` služi za poziv potprograma. Ona zapravo trenutnu vrijednost programskog brojila postavlja na stog (trenutna vrijednost programskog brojila – sadržaj registra `eip`, adresa instrukcije iza `call` instrukcije, tj. adresa instrukcije `cli`). Potom, u programsko brojilo se stavlja zadana adresa funkcije (`print_hello`). Iduća će instrukcija zato biti prva od zadane funkcije (obavljen je skok u funkciju). Za povratak iz potprograma se koristi instrukcija `ret` koja uzima podatak s vrha stoga i stavlja ga u programsko brojilo (instrukcija se ne pojavljuje izravno u ovom kôdu, ona nastaje prevođenjem kôda iz `hello.c` datoteke). Ako se ispravno rukuje sa stogom na njegovu vrhu će se u tom trenutku naći povratna adresa, prethodno spremljena `call` instrukcijom.

Odradom sveg zadanog, sustav treba zaustaviti. To se može i beskonačnom petljom. Ovdje je u petlji dodana instrukcija `hlt` – instrukcija koja zaustavlja rad procesora do pojave prekida. S obzirom na to da je `hlt` u petlji, bez obzira na moguće prekide program neće ići dalje. U ovom je slučaju prihvat prekida cijelo vrijeme onemogućen (zastavicom u registru stanja te dodatno instrukcijom `cli`) te petlja i nije potrebna (ali kasnije jest).

Isječak kôda 4.9. Chapter_01_Startup/01_Startup/i386/startup.S

```
45 .section .bss
46 .align  4
47
48         .lcomm _stack_, 4096
49 stack:
```

Na kraju datoteke definira se početak novog odjeljka `.bss`, poravnatog na 4 okteta, u kojem se zauzima 4096 okteta. Početak tog bloka označen je labelom `_stack_`, a kraj labelom `stack`. Odjeljak `.bss` se koristi za podatke koji nisu početno inicijalizirani ili trebaju biti inicijalizirani nulama te zato ne trebaju zauzimati mjesto na mediju s kojeg se program učitava, već se pri učitavanju u spremniku može *tada* zauzeti potreban spremnički prostor.

4.3. Datoteka `hello.c`

U datoteci `hello.c` nalazi se jedina "korisna" funkcija – funkcija `print_hello`.

Isječak kôda 4.10. Chapter_01_Startup/01_Startup/i386/hello.c

```
1 /*! Print on console using video memory */
2
3 #define VIDEO   ((volatile char *) 0x000B8000) /* address of video memory  */
4 #define COLS    80 /* number of characters in a column */
5 #define ROWS    25 /* number of characters in a row */
6 #define ATTR    7  /* font: white char on black bacground */
```

Za ispisivanje na zaslon koristi se izravno upisivanje u spremnički prostor dodijeljen grafičkoj kartici (u *video memoriju*, za osnovni način rada). Zaslon se sastoji od 24 retka po 80 stupaca (znakova). Za upis nekog znaka na određenu lokaciju dovoljno je upisati ASCII kôd na odgovarajuće mjesto te postaviti atribute tog znaka (boju, svjetlinu, način prikaza) na susjednu spremničku lokaciju. Na primjer, za zapis znaka `'A'` na prvo mjesto u gornjem lijevom uglu zaslona, na adresu `0xB8000` upisuje se `'A'`, a na susjedno (adresu `0xB8001`) atribut `7` (za uobičajeni prikaz bijelog znaka na crnoj pozadini).

Isječak kôda 4.11. Chapter_01_Startup/01_Startup/i386/hello.c

```
8  /*! Print "Hello world!" */
9  void print_hello()
10 {
11         int i;
12         char hello[] = "Hello World!";
13         volatile char *video = VIDEO;
14
15         /* erase screen (set blank screen) */
16         for (i = 0; i < COLS * ROWS; i++)
17                 video[ i * 2 ] = video[ i * 2 + 1 ] = 0;
18
19         /* print "Hello World!" on first line */
20         for (i = 0; i < 13; i++)
21         {
22                 video[ i * 2 ]     = hello[i];
23                 video[ i * 2 + 1 ] = ATTR;
24         }
25
26         return;
27 }
```

Gornja funkcija najprije obriše cijeli zaslon (stavlja nulu na svako mjesto i za znak i za atribut znaka) te potom u prvu liniju upiše poruku `Hello World`.

4.4. Primjer sata

Ispis poruke na zaslon je možda jedna od najtrivijalnijih stvari. Međutim, na sličan se način mogu ostvariti i složeniji sustavi kod kojih se u jednom programu upravlja svim komponentama sustava. Upravo stoga je u direktoriju `Chapter_01_Startup/02_Example_clock` dodan nešto složeniji primjer – primjer sa satom. U ovom se programu, osim što se izravno upisuje u spremnik grafičke kartice, koristi i brojilo sklopa *Intel 8253*. Način rada navedenog brojila detaljnije je opisan u poglavlju 8.1. Ukratko, brojilo se programira da odbrojava od zadane vrijednosti do nule te opet od zadane vrijednosti. Kad dođe do nule može izazvati prekid, ali se to ovdje ne koristi, već se u petlji čekalici pričeka jedan ciklus brojenja (da se ponovno učita zadana vrijednost).

Sat uključuje prikaz sati (0-23), minuta (0-59), sekundi (0-59) te stotinki (0-99). Pri pokretanju započinje s vremenom 00:00:00:00. Zbog značajnog trajanja izvođenja ulazno-izlaznih operacija, kao i zbog emulatora, moguća su (primjetna) odstupanja od očekivanog protoka vremena.

Za vrlo jednostavne ugrađene sustave nije potrebno izgrađivati operacijski sustav ili složenije podsustave – načelo jednostavne upravljačke petlje je često dovoljno.

Pitanja za vježbu 4

1. Kako se pokreće operacijski sustav? Koja je zadaća BIOS-a?
2. Koje poslove obavlja *gcc*, a koje *ld* pri izgradnji slike sustava (`.elf` datoteke)?
3. Što je potrebno napraviti u dijelu programa koji preuzima upravljanje sustavom?
4. Kako se iz asemblera pozivaju funkcije ostvarene u C-u? Što sve treba prethodno napraviti? Mogu li se iz C-a pozivati funkcije ostvarene u asembleru? Kako to napraviti?

5. Organizacija kôda

Kôd prikazan u prethodnom poglavlju sastoji se samo od dvije datoteke. To, međutim, neće biti dovoljno za složenije sustave.

Pri razmatranju i izgradnji se složeni sustavi na razne načine nastoje pojednostaviti . Osnovno načelo pojednostavljenja je *podijeli i vladaj*. Složene se aplikacije, sustavi i komponente dijele u nekoliko *slojeva* ili *razina*. Primjerice, mrežni se podsustav u razmatranjima dijeli na 7 slojeva (*OSI-RM*), dok u praksi ipak samo na 5 slojeva.

Operacijski sustav se može podijeliti na razne načine. Jedan od uobičajenih je podjela na slojeve (gdje se složene operacije rastavljaju na jednostavnije, u zasebnim slojevima), a drugi podjela na komponente (zasebne komponente odrađuju različite zadaće/funkcije u sustavu) koje nazivamo *podsustavima*. Najčešće korišteni podsustavi su: upravljanje spremnikom, upravljanje ulazno-izlaznim napravama, upravljanje dretvama i procesima, datotečni podsustav i mrežni podsustav.

U podjeli na slojeve možemo izdvojiti nekoliko slojeva. Najniži sloj – sloj arhitekture, omogućuje iskorištenje sklopovlja (upravljački programi, posebnosti procesora i slično), tj. on apstrahira sklopovlje (engl. *hardware abstraction layer*). Iznad njega, nalazi se sloj koji obavlja najosnovnije operacije – sloj jezgre. Složenije operacije potrebne višoj razini, a koje se ostvaruju korištenjem jezgre, nalaze se u idućem sloju – sloju sustavskih funkcija. Kao najviši sloj, iznad operacijskog sustava, nalaze se programi koji izvode za korisnika korisne operacije, pritom koristeći sučelje operacijskog sustava.

Podjela u podsustave i slojeve može biti ostvarena samo statički, u okviru izvornih kôdova ili i dinamički, vidljiva prilikom rada sustava.

Nastojeći pojednostaviti razumijevanje, razvoj i prenosivost, prikazani sustav[1] koristi podjelu u slojeve i podsustave, prema slici 3.1. S obzirom na to da su podsustavi vrlo jednostavni, ostvareni su u zasebnim datotekama. Različiti slojevi nalaze se u različitim direktorijima te se i samom njihovom strukturom odvajaju:

- sloj arhitekture – *arch* – kôd vezan uz samo sklopovlje (arhitekturu)
- sloj jezgre – *kernel* – kôd jezgre operacijskog sustava
- sloj sustavskih funkcija – *api* – kôd sustavskih funkcija (*API*) koje koriste programi
- sloj programa — *programs* – kôd korisničkih programa
- sloj biblioteka — *lib* – kôd pomoćnih funkcija koje se koriste iz ostalih slojeva.

Sučelje bi trebalo odvojiti programe od jezgre. Programi trebaju poznavati samo sučelje, a ne i način i pojedinosti ostvarenja. U prvim je koracima sučelje samo skup deklaracija koji prevode *pozive jezgre* (u idućim koracima to više neće biti tako).

Iako se sve za sada izvodi u tzv. *jezgrenom načinu rada* (nadglednom, povlaštenom) dobro je odvojiti programe od jezgre i na razini izvornih kôdova. Raznim mehanizmima programskog jezika C to odvajanje se nastoji očuvati.

Slika 5.1. prikazuje novu struktura kôda (u inkrementu 02/01), uključujući direktorije i datoteke.

U početnom direktoriju (`01_Source_tree`) ostaje samo datoteka `Makefile`. Ostatak kôda raspoređen je u direktorije kako je prikazano na slici 5.1. Svaki sloj je u vlastitom direktoriju:

[1] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_02_Startup/01_Source_tree`.

```
arch
    i386
        config.ini
        types.h
        descriptor.h
        descriptor.c
        processor.h
        io.h
        boot
            startup.S
            ldscript.ld
        drivers
            print.c
    arm
        (slično kao i386)
kernel
    startup.c
    kprint.h
    kprint.c
api
    stdio.c
lib
    string.c
include
    arch
        types.h
        processor.h
        multiboot.h
        print.h
    kernel
        kprint.h
    api
        prog_info.h
        stdio.h
    lib
        string.h
programs
    hello_world
        hello_world.c
Makefile
```

Slika 5.1. Izvorni kôdovi: direktoriji i datoteke za 02/01

`arch`, `kernel`, `api`, `programs` te `lib`. Sučelja slojeva su deklarirana u direktoriju `include` i to zasebno za svaki sloj. Način prevođenja definiran je datotekama `Makefile` i `config.ini` (iz *arch* sloja odabrane arhitekture).

Sloj arhitekture – *arch*

Funkcije u sloju *arch* imaju prefiks `arch_`, dok funkcije u sloju jezgre imaju prefiks `k_` (ponegdje i samo `k`). Gotovo svaka `.c` datoteka ima svoje zaglavlje (`.h` datoteku) u kojem su deklarirane funkcije koje se koriste u `.c` datoteci, razne konstante i strukture podataka. Ponekad nisu sve funkcije potrebne izvan te datoteke te su one namjerno i skrivene[2].

Sloj *arch* je projektiran s namjerom da bude zamjenjiv. U direktorij `arch` treba postaviti ostvarenje sloja za razne arhitekture. Trenutno je ostvarena podrška za arhitekture i386 (`i386`) i ARM (`arm`).

U sloju *arch* preuzima se kontrola nad sustavom (`i386/boot/startup.S`) te se ona prosljeđuje jezgri preko funkcije `k_startup` (`kernel/startup.c`). Konstante koje trebaju radi *multiboot* zaglavlja definirane su u `include/arch/multiboot.h`. Ostatak `i386/boot/startup.S` datoteke sličan je onome iz prvog inkrementa, uz sitne razlike.

U `i386/boot` direktoriju nalazi se datoteka `ldscript.ld` koja je opisana u idućem odjeljku.

Nadalje, u sloju *arch*, u datoteci `i386/drivers/print.c` ostvarena je funkcija za ispis niza znakova na zaslon korištenjem izravnog upisa u spremnički prostor grafičke kartice. U `include/arch/print.h`, kao i svim zaglavljima, na početku se nalazi zaštita od višestrukog uključivanja:

```
#pragma once
```

Tu zaštitu je moguće napraviti i na drugi način s naredbom `#ifndef`:

[2]Iako se prevode samo `.c` datoteke, prilikom prevođenja se interno u prevoditelju u zadanu datoteku učitavaju zaglavlja na mjestima gdje su navedena: `#include <nesto.h>` se zamjenjuje sa sadržajem datoteke `nesto.h`. Više o korištenju zaglavlja u dodatku C.

```
#ifndef _ARCH_PRINT_H_
#define _ARCH_PRINT_H_
... /* sadržaj datoteke */
#endif      /* _ARCH_PRINT_H_ */
```

Za svaku bi datoteku trebalo definirati vlastite varijable, primjerice prema imenu datoteke, kao u gornjem slučaju, uz dodatak sloja/direktorija, tako da ime bude jedinstveno za tu datoteku.

Ostatak datoteke `i386/drivers/print.c` je vrlo sličan već prikazanome u prvom inkrementu (`hello.c`). Razlika je u korištenju pomoćne funkcije `memset` umjesto izravnog korištenja petlje. Datoteka `arch/types.h` definira osnovne tipove podataka, tj. povezuje kraće oznake s uobičajenim tipovima, primjerice definira tip `int32` kao 32-bitovni cijeli broj.

Sloj jezgre – *kernel*

U sloju jezgre (`kernel`) nalaze se datoteke: `startup.c` i `kprint.c`. Prava početna funkcija sustava iz koje se sve pokreće je `k_startup` (jedina funkcija datoteke `kernel/startup.c`). U `kernel/kprint.c` nalaze se funkcija za inicijalizaciju zaslona `kconsole_init` te za ispis niza znakova na zaslon `kconsole_print_word`. Obje funkcije koriste ekvivalentne funkcije sloja *arch* (definiranih u `include/arch/print.h` i ostvarenih u `i386/drivers/print.c`). Sučelje koje jezgra nudi definirano je u `include/kernel`. Trenutno je tamo samo jedna datoteka `kprint.h` koja definira sučelje koje se nudi višem sloju – sloju *api*.

Sloj sustavskih funkcija – *api*

Sloj *api* sadržan je u direktoriju `api`. U ovom inkrementu ovaj sloj sadržava samo jednu datoteku `stdio.c` koja ostvaruje funkcije `console_init` i `console_print_word` koje mogu koristiti programi. Te su funkcije definirane, kao sučelje *api* sloja, u `include/api/stdio.h`. U istom direktoriju (`include/api`) nalazi se i datoteka `prog_info.h` u kojoj se nalaze deklaracije svih programa (trenutno samo program `hello_world`) koji se mogu pokrenuti iz jezgre (`kernel/startup.c`).

Sloj programa – *programs*

Programi se nalaze u direktoriju `programs` u poddirektoriju koji odgovara imenu programa. U direktoriju `programs` trenutno se nalazi samo direktorij `hello_world` s istoimenom datotekom `hello_world.c` i unutar nje istoimenom funkcijom `hello_world()` koja ispisuje poruku (`Hello World!`).

Sloj biblioteka – *lib*

Direktorij `lib` sadržava pomoćne funkcije za rad sa spremnikom i nizom znakova (`string.c`). Kasnije se tu pojavljuju i operacije za rad s listom i algoritmi za dinamičko upravljanje spremnikom. Sučelja operacija koje su ovdje ostvarene deklarirana su u `include/lib` direktoriju, u odgovarajućim datotekama.

Sučelja slojeva – *include*

Sučelja svih slojeva deklarirana su u odgovarajućim datotekama direktorija `include`. U direktoriju `include/types` nalaze se definicije dodatnih tipova podataka i konstanti namijenjene za korištenje uglavnom iz programa.

Isječak kôda 5.1. Chapter_02_Source_tree/01_Source_tree/include/types/basic.h

```
4  #include <arch/types.h>
5
6  #if __WORD_SIZE >= 32
7
8  #ifndef MEM_TEST
9  typedef word_t  size_t;
```

```
10 typedef sword_t ssize_t;
11 #endif /* MEM_TEST */
12
13 #else /* size_t must be 32 bits or more */
14
15 typedef uint32  size_t;
16 typedef int32   ssize_t;
17
18 #endif /* __WORD_SIZE */
19
20 #define NULL            ((void *) 0)
21
22 #define FALSE           0
23 #define TRUE            0x0f
```

Definicije tipova koji mogu biti različite duljine na različitim arhitekturama preuzimaju se iz sloja *arch* (odgovarajućih `.h` datoteka).

5.1. Datoteka `Makefile`

Prevođenje ovako "razgranatog" kôda moglo bi biti problematično da ne postoje jednostavni alati za automatiziranje prevođenja. U početnom inkrementu koristila se skripta ljuske, tj. datoteka `build.sh`, u kojoj su ručno navedene sve naredbe. U svim ostalim koracima koristi se alat *Make* i definicija prevođenja zadana u datoteci `Makefile`.

5.1.1. Primjer prevođenja

Neka u primjeru postoji pet datoteka: `a.h`, `a.c`, `b.h`, `b.c` te `brojke.c` sa sljedećim sadržajima.

a.h

```
#define A 5
double a ( double p );
```

b.h

```
#define B (3.14/5)
double b ( double p );
```

a.c

```
#include <math.h>
#include "a.h"
#include "b.h"

double a ( double p )
{
   return A * cos ( b(p) );
}
```

b.c

```
#include "b.h"

double b ( double p )
{
   return B * p;
}
```

brojke.c

```
#include <stdio.h>
#include "a.h"

int main ()
{
   printf ( "Neki broj = %g\n", a(33) );

   return 0;
}
```

"Ručno" prevođenje treba napraviti naredbom:

```
gcc a.c b.c brojke.c -lm -o brojke
```

Iako na prvi pogled izgleda jednostavno, s tipkovnice je potrebno unijeti 34 znaka. Kad bi broj datoteka bio veći to bi se povećalo. Navedeno bi trebalo utipkati svaki put kada se nešto promijeni!

Da bi se ubrzao postupak prevođenja uobičajeno je da se koriste razna pomagala. Jedno od njih je korištenje alata *Make* i pripadajuće datoteke s postavkama: `Makefile`. Za zadani sustav izvornih datoteka, `Makefile` bi mogao biti:

```
brojke: a.o b.o brojke.o
    gcc a.o b.o brojke.o -lm -o brojke

a.o: a.c a.h b.h
    gcc -c a.c

b.o: b.c b.h
    @gcc -c b.c

brojke.o: brojke.c a.h
    gcc -c brojke.c
```

Osnovni (prvi navedeni) "predmet" prevođenja (engl. *target*) je `brojke`. Kada se pokrene naredba `make` bez parametara pokreće se izgradnja prvog predmeta prevođenja, pretpostavljenog za izradu. S desne strane znaka ":" svakog predmeta su datoteke (ili drugi predmeti prevođenja) o kojima je predmet ovisan. U liniji ispod (koja je "uvučena") su upute kako ga izgraditi. Ostali predmeti prevođenja `a.o`, `b.o` i `brojke.o` slijede slično načelo. Jedan predmet može u ovisnostima imati i druge predmete. U tom slučaju, pri zradi tog predmeta se najprije pokreće izrada tih drugih predmeta, a tek po njihovu dovršetku početni predmet. U gornjem primjeru će se pri izradi predmeta `brojke` najprije napraviti predmeti `a.o`, `b.o` i `brojke.o`.

U liniji ispod predmeta, "uputama kako ga izraditi", se najčešće pokreće prevođenje ili povezivanje. Prevođenje se u gornjem slučaju obavlja za predmete `a.o`, `b.o` te `brojke.o` (s `gcc -c`). Povezivanje već prevedenih datoteka u izvršnu (izlaznu) datoteku radi se samo kod predmeta `brojke` (`gcc a.o b.o brojke.o -lm -o brojke`).

Znak `@` ispred naredbe `gcc` nalaže alatu *Make* da pri pokretanju naredbe istu ne ispiše na zaslon – ispisat će se samo rezultati pokretanja, tj. ono što sam *gcc* ispiše pri prevođenju (poruke upozorenja i greške).

Prevođenje korištenjem alata *Make* obavlja se upisivanjem naredbe `make`.

```
$ make
gcc -c a.c
gcc -c brojke.c
gcc a.o b.o brojke.o -lm -o brojke
```

Poruka (`gcc -c b.c`) neće se ispisati zbog znaka `@` ispred `gcc` pri prevođenju datoteke `b.c`.

Alat *Make* prati vremena promjena datoteka i pri prevođenju prevodi samo one koje je potrebno prevesti, tj. koje su se promijenile od zadnjeg prevođenja. U gornjem primjeru, ako se nakon početnog prevođenja promijeni samo datoteka `brojke.c` samo će se ona prevesti te će se ponovno obaviti i povezivanje. U gornjem primjeru broj datoteka je mali te čak i kad bismo svaki put sve datoteke ponovno prevodili ne bismo puno vremena izgubili. Međutim, u projektima koji imaju jako puno datoteka prevođenje svega može znatno potrajati. Ipak, ako je promjena nastala samo u jednoj datoteci s izvornim kôdom nije sve potrebno prevoditi već samo tu datoteku. Time se postupak prevođenja znatno ubrzava. To je i jedno od glavnih razloga korištenja datoteke `Makefile` i sličnih datoteka.

5.1.1.1. Implicitna pravila

Za prikazani primjer datoteka `Makefile` se može i pojednostaviti. Naime, alat *Make* može i sam zaključiti neke stvari. Primjerice, da bi dobio datoteku `a.o` treba prevesti datoteku `a.c`

kada takva postoji. Stoga predmeti `a.o`, `b.o` i `brojke.o` i nisu neophodni (ne uvijek, ali više o tome kasnije). Predmeti koji nisu navedeni u `Makefile`-u, ali ih *Make* sam svejedno izgrađuje izvode se prema takozvanim "implicitnim pravilima". Implicitno pravilo se može iskoristiti na način da se neki predmeti prevođenja u potpunosti izostave (kao npr. za `a.o`) ili može biti zadana samo definicija predmeta (prva linija prema `naziv_predmeta: ovisnosti`). Da bi se implicitna pravila mogla što više koristiti dodatno se koriste "implicitne varijable" preko kojih se prevoditelju i povezivaču mogu prenijeti zastavice i ostale naredbe. Pri prevođenju se kod implicitnih pravila koristi varijabla `CFLAGS`, a pri povezivanju varijable `LDFLAGS` i `LDLIBS`.

Prethodni `Makefile` bi se primjenom implicitnih pravila mogao napisati na sljedeći način.

```
CFLAGS = -MMD # opisano kasnije
LDFLAGS = -O # optimiraj kod; dodano samo radi prikaza položaja
LDLIBS = -lm

brojke: brojke.o a.o b.o # brojke.o maknut na prvo mjesto!

-include *.d # opisano kasnije
```

Prevođenje s alatom *Make* pokrenuti će naredbe:

```
$ make
cc -MMD   -c -o brojke.o brojke.c
cc -MMD   -c -o a.o a.c
cc -MMD   -c -o b.o b.c
cc -O   brojke.o a.o b.o  -lm -o brojke
```

Program `cc` je zapravo poveznica (link) na `gcc`. Ukoliko bi se želio definirati prevoditelj za implicitna pravila, to se može preko implicitne varijable `CC` (npr. dodati liniju `CC = gcc` u `Makefile`).

Kod implicitnih pravila, za prevođenje *Make* pokreće naredbe oblika:

```
$(CC) $(CFLAGS) -c -o datoteka.o datoteka.c
```

a za povezivanje naredbu:

```
$(CC) $(LDFLAGS) "sve potrebne .o datoteke" $(LDLIBS) -o program
```

5.1.1.2. Smještaj međurezultata prevođenja

Pri prevođenju (*kompajliranju*) projekata koriste se razni pristupi o smještaju prevedenih datoteka (`.o` datoteka).

Najjednostavniji pristup ostavlja prevedene datoteke uz izvorne (u istim direktorijima). Ovaj pristup nije korišten u projektu Benu iz razloga što stvara “gužvu” u direktorijima s izvornim kôdovima te se smanjuje preglednost.

Po drugom pristupu sve se prevedene datoteke smještaju u jedan zaseban direktorij. Problem s ovim pristupom je u imenima datoteka: ako postoje dvije datoteke istog imena u različitim direktorijima onda će prevođenje druge prebrisati prvu prevedenu. Tome se može doskočiti tako da se ime prevedene datoteke proširi sa cijelom putanjom do datoteke. Na primjer, ako se u direktorijima `kernel` i `arch` nalaze datoteke istog imena: `print.c` tada bi se prevedene datoteke mogle nazvati redom `kernel_print.o` te `arch_print.o`.

Prema trećem pristupu u svaki se direktorij s izvornim kôdom postavlja datoteka `Makefile` koja sadrži upute kako te datoteke prevesti. Iz početne datoteke `Makefile` se tada spušta po direktorijima i pokreće lokalno prevođenje. Ovaj pristup omogućava znatno više lokalnog podešavanja pri prevođenju i često se koristi za veće projekte. Primjeri 5.1. i 5.2. prikazuju takvo korištenje.

Primjer 5.1. Makefile u svakom direktoriju s kodom(1)

Neka se u nekom sustavu nalaze direktoriji i datoteke prikazane na lijevom dijelu slike 5.2. U svakom direktoriju nalazi se zasebni Makefile koji definira prevođenje datoteka iz tog direktorija te pozivanje istog postupka za poddirektorije. Objekti koji nastaju prevođenjem i povezivanjem navedeni su s desne strane slike. Glavni cilj prevođenja jest program koji izgrađuje početni Makefile nakon što su prethodno izgrađeni svi ostali objekti.

datoteke	**objekti nastali prevođenjem**
Makefile	=> **program** = main.o +zaprimi.o + obradi.o + ispisi.o
main.c	=> main.o
zaprimi	
Makefile	zaprimi.o = zaprimi.o
zaprimi.c	zaprimi.c => zaprimi.o
zaprimi.h	
obradi	
Makefile	obradi.o = obradi1.o + obradi2.o
obradi1.c	obradi1.c => obradi1.o
obradi2.c	obradi2.c => obradi2.o
obradi.h	
obradi2.h	
ispisi	
Makefile	ispisi.o = ispisi.o
ispisi.c	ispisi.c => ispisi.o
ispisi.h	

Slika 5.2. Direktoriji, datoteke, objekti

Svaki Makefile u nekom direktoriju izgrađuje "modul" za taj direktorij korištenjem objekata nastalih prevođenjem datoteka iz tog direktorija i direktorija ispod (poddirektorija). Početni Makefile brine o datoteci main.c te rekurzivno poziva *make* nad poddirektorijima. Sadržaj datoteke Makefile u početnom direktoriju naveden je u nastavku.

Isječak kôda 5.2. Početni Makefile

```
CFLAGS = -Iispisi -Iobradi -Izaprimi
LDFLAGS = -O2
LDLIBS = -lm

program: main.o zaprimi/zaprimi.o obradi/obradi.o ispisi/ispisi.o
    gcc $(LDFLAGS) main.o zaprimi/zaprimi.o obradi/obradi.o ispisi/ispisi.o \
    $(LDLIBS) -o program

.PHONY: zaprimi/zaprimi.o obradi/obradi.o ispisi/ispisi.o
zaprimi/zaprimi.o:
    make -C zaprimi
obradi/obradi.o:
    make -C obradi
ispisi/ispisi.o:
    make -C ispisi

.PHONY: brisi
brisi:
    make -C zaprimi brisi
    make -C obradi brisi
    make -C ispisi brisi
    -rm -f main.o program

.PHONY: pokreni
pokreni: program
    ./program
```

Na početku je definirana varijabla `CFLAGS` koja se koristi pri implicitnom prevođenju (kada nije navedeno kako iz .c dobiti .o). U ovom primjeru varijabla sadrži upute prevoditelju gdje da dodatno traži zaglavlja pri prevođenju (zastavica `-I<direktorij>`). Naime, u `main.c` koristi se `#include <zaprimi.h>` gdje je ime zaglavlja stavljeno unutar <>. Time prevoditelj mora potražiti to zaglavlje među svim direktorijima koji su mu označeni da sadrže zaglavlja.

Varijable `LDFLAGS` i `LDLIBS` bi se koristile u implicitnom pravilu za povezivanje. Međutim, obzirom na imena datoteka *make* ne zna kako generirati `program` te su za to navedena eksplicitna pravila u kojima se koriste te varijable (eksplicitno).

Objekte `zaprimi/zaprimi.o`, `obradi/obradi.o` i `ispisi/ispisi.o` ne generira početni `Makefile` već za to koristi rekurzivne pozive u tim direktorijima (npr. `make -C zaprimi`).

Predmeti označeni s `.PHONY` nalažu alatu *make* da svaki puta izvede operacije u tijelu tih predmeta, ne provjeravajući "ažurnost" tih objekata na druge načine. Najčešće takvi predmeti i nemaju zadane datoteke s desne strane dvotočke (kao predmet `brisi`).

Datoteka `Makefile` u direktoriju `zaprimi` definira predmet `zaprimi.o` i datoteke o kojima ovisi te predmet za brisanje suvišnih datoteka koje nastaju tim prevođenjem. Kao i za sva prevođenja koristi se implicitno pravilo (nije navedeno kako se iz .c dolazi do .o).

Isječak kôda 5.3. `Makefile` u `zaprimi`

```
zaprimi.o: zaprimi.c zaprimi.h

.PHONY: brisi
brisi:
    -rm -f zaprimi.o
```

Datoteka `Makefile` u direktoriju `obradi` definira predmet `obradi.o` koji nastaje prevođenjem dviju datoteka i njihovim povezivanjem. Obzirom da pri tom povezivanju ne nastaje program već samo još jedan datoteka koja objedinjava prethodne dvije koristi se program *ld* sa zastavicom `-r`.

Isječak kôda 5.4. `Makefile` u `obradi`

```
obradi.o: obradi1.o obradi2.o
    ld -r obradi1.o obradi2.o -o obradi.o

obradi1.o: obradi1.c obradi2.h

obradi2.o: obradi2.c obradi2.h

.PHONY: brisi
brisi:
    -rm -f obradi.o obradi1.o obradi2.o
```

Datoteka `Makefile` u direktoriju `ispisi` slično kao i `Makefile` u direktoriju `zaprimi` prevodi samo jednu datoteku. Međutim, za to prevođenje potrebno je postaviti zastavicu `-DDIR=\"ISPISI\"` obzirom da se ime `ISPISI` pojavljuje u kodu `ispisi.c` u liniji: `printf("%g (iz direktorija "DIR ")\n", y);`.

Isječak kôda 5.5. `Makefile` u `ispisi`

```
CFLAGS = -DDIR=\"ISPISI\"

ispisi.o: ispisi.c ispisi.h

.PHONY: brisi
brisi:
```

```
    -rm -f ispisi.o
```

Primjer 5.2. Makefile u svakom direktoriju s kodom (2)(info)

Neka se u nekom sustavu nalaze direktoriji i datoteke prikazane na lijevom dijelu slike 5.3. U svakom direktoriju nalazi se zasebni Makefile koji definira prevođenje datoteka iz tog direktorija te pozivanje istog postupka za poddirektorije. Objekti koji nastaju prevođenjem i povezivanjem navedeni su s desne strane slike. Glavni cilj prevođenja jest program koji izgrađuje početni Makefile nakon što su prethodno izgrađeni svi ostali objekti.

datoteke	objekti nastali prevođenjem
config.ini	
Makefile	=> **program** = main.o + mod-A.o + mod-B.o
main.c	=> main.o
include	
postavke.h	
A	=> mod-A.o = A1.o + A2.o + a/mod-a.o + b/mod-b.o
A1.c	=> A1.o
A2.c	=> A2.o
Makefile	
a	=> mod-a.o = a1.o + i/mod-i.o + ii/mod-ii.o
a1.c	=> a1.0
Makefile	
i	=> mod-i.o = i1.o + i2.o
i1.c	=> i1.o
i2.c	=> i2.o
Makefile	
ii	=> mod-ii.o = ii1.o
ii1.c	=> ii1.o
Makefile	
b	=> mod-b.o = b1.o
b1.c	=> b1.o
Makefile	
B	=> mod-B.o = B1.o
B1.c	=> B1.o
Makefile	

Slika 5.3. Direktoriji, datoteke, objekti

Svaki Makefile u nekom direktoriju izgrađuje "modul" za taj direktorij korištenjem objekata nastalih prevođenjem datoteka iz tog direktorija i direktorija ispod (poddirektorija). Primjer sadržaja datoteke Makefile u direktoriju a prikazuje kod 5.6.

Isječak kôda 5.6. Makefile u direktoriju A/a

```
include $(POCETNI_DIREKTORIJ)/config.ini

VAR1 = 11 # redefiniranje vrijednosti za varijable iz config.ini
VAR2 = 2
CFLAGS += -D VAR1=$(VAR1) -D VAR2=$(VAR2)

OBJEKTI = a1.o  # objekti koje izgrađuje ova skripta
MODULI = i/mod-i.o ii/mod-ii.o # objekti iz poddirektorija
MODUL = mod-a.o # objekt koji je konačan rezultat,
                # koji se koristi iz viših direktorija

# spajanje više objekata u jedan, u "modul"
$(MODUL): $(OBJEKTI) $(MODULI)
```

```
    $(LD) -r $^ $(LDFLAGS) -o $@

# prevođenje datoteka iz ovog direktorija
a1.o: a1.c $(POCETNI_DIREKTORIJ)/include/postavke.h
    $(CC) $(CFLAGS) -o $@ -c $<

.PHONY: $(MODULI) # oznaka predmeta koje uvijek treba izgraditi
                  # neka Makefile iz tih direktorija brine o izgradnji

# prevođenje datoteka iz svih poddirektorija
i/mod-i.o:
    $(MAKE) -C i  # pozovi "make" za poddirektorij "i"
ii/mod-i.o:
    $(MAKE) -C ii # pozovi "make" za poddirektorij "ii"

# brisanje
.PHONY: brisi
brisi:
    $(MAKE) -C i  brisi
    $(MAKE) -C ii brisi
    rm -f $(OBJEKTI) $(MODUL)
```

Prethodni pristup traži dosta ručnih podešavanja i mnoštvo `Makefile` datoteka te nije korišten u Benu.

5.1.1.3. Prevođenje korišteno u Benu

Odabrani pristup u projektu će u zasebnom direktoriju (nazvanom `build`) napraviti identično stablo direktorija kao za izvorne datoteke te će se pri prevođenju prevedene datoteke smjestiti u ekvivalentne direktorije. Na primjer, za datoteku `kprint.c` iz direktorija `kernel`, prevedena datoteka `print.o` postavit će se u direktorij `build/kernel`, dok će se rezultat prevođenja datoteke `print.c` iz direktorija `arch/i386/drivers` staviti u `build/arch/i386/drivers`. Na ovaj način ništa se ne mijenja u direktorijima s izvornim kôdovima. Dodatno, asemblerske datoteke (koje završavaju sa `.S`) prevode se u objektne datoteke sa sufiksom `.asm.o` tako da se u istom direktoriju s izvornim kôdovima istovremeno mogu naći i `.c` i `.S` datoteke istog imena (tj. početnog dijela prije ekstenzije).

S obzirom na to da trenutno razne datoteke ne traže posebnu pažnju (posebne postavke/zastavice pri prevođenju), u datoteci `Makefile` nisu pojedinačno navedene datoteke već samo direktoriji u kojima se datoteke s izvornim kôdom nalaze. Alat *Make* će prema uputi pretražiti te direktorije i odrediti datoteke za prevođenje (i prevesti ih). Dodavanje nove izvorne datoteke u postojeći direktorij ne zahtijeva izmjenu datoteke `Makefile`, ali dodavanje nove datoteke u novi direktorij zahtijeva dodavanje tog direktorija u popis onih koji se pretražuju.

Kao što je već rečeno, *Make* pri pokretanju prevodi samo one datoteke koje je potrebno: ili u odredištu ne postoji izlazna (prevedena) datoteka ili je vrijeme promjene izvorne datoteke (`.c`) novije od vremena stvaranja ili promjene postojeće izlazne datoteke (`.o`). Ponekad treba prevesti datoteke i ako se nisu mijenjale, ali su se mijenjala zaglavlja koja one uključuju. Ovisnosti pojedinih datoteka o pojedinim zaglavljima može se zadati ručno u datoteci `Makefile`. Međutim, za projekte s većim brojem datoteka to postaje zamorno i teško za održavanje. Prevoditelj *gcc* može pomoći te s odgovarajućim zastavicama može sam generirati ovisnosti. Generirane se ovisnosti spremaju u zasebne datoteke, uz izlazne datoteke, sa sufiksom `.d`. Na primjer, za datoteku `kernel/kprint.c` generira se i datoteka `build/kernel/print.d`. Te se `.d` datoteke učitavaju na kraj datoteke `Makefile` pri pokretanju idućeg prevođenja. Prvo pokretanje izgradnje sustava će uvijek prevoditi svaku datoteku s obzirom na to da njen objekt ne postoji. Na taj način će se prvi puta generirati odgovarajuće `.d` datoteke koje će se onda koristiti u idućim izgradnjama za provjeru treba li neku datoteku ponovno prevesti ili ne s obzirom na njene ovisnosti (je li se mijenjala neka `.h` datoteka koju ona uključuje).

Nakon dovršetka prevođenja sadržaj direktorija `build` izgleda kao na slici 5.4.

```
arch
    i386
        descriptor.o
        descriptor.d
        boot
            startup.o
            startup.d
        drivers
            print.o
            print.d
kernel
    startup.o
    startup.d
    kprint.o
    kprint.d
api
    stdio.o
    stdio.d
programs
    hello_world
        hello_world.o
        hello_world.d
lib
    string.o
    string.d
ARCH = link na arch/i386
ldscipt.ld
01_Source_tree.elf
```

Slika 5.4. Direktoriji i datoteke nastale prevođenjem (`build` direktorij)

U isti odredišni direktorij `build` smjestit će se i slika sustava (`.elf` datoteka). Kao što se vidi iz priloženog, svaka `.c` i `.S` datoteka preslikava se u `.o` i `.d` datoteku (s dodatkom `.asm` za `.S` datoteke). Sve se `.o` datoteke potom povežu u `01_Source_tree.elf` što je konačni rezultat prevođenja – slika sustava koji se može pokrenuti u emulatoru.

Sve navedene radnje definirane su u datotekama `Makefile` te `config.ini` koja se uključuje u prvu (s `include $(CONFIG_INI)` na početku datoteke `Makefile`) i zapravo je samo proširenje prve radi odvajanja postavki (`config.ini`) od operacija (`Makefile`). U nastavku se te datoteke detaljnije analiziraju.

Na početku datoteke definirane su varijable koje se koriste u nastavku.

Isječak kôda 5.7. Chapter_02_Source_tree/01_Source_tree/arch/i386/config.ini

```
3  # Common configuration
4  #-----------------------------------------------------------------------------
5  OS_NAME = "Benu"
6  NAME_MAJOR := $(shell basename "`cd ..; pwd -P`")
7  NAME_MINOR := $(shell basename "`pwd -P`")
8  PROJECT := $(NAME_MINOR)
9
10 ARCH ?= i386
11 VERSION = 1.0
```

Komentar se u datoteci `Makefile` označava znakom `#` – sve što je iza njega pa do kraja linije je komentar i *Make* ga preskače (ne obrađuje).

U prvom dijelu datoteke nalaze se definicije varijabli koje se prenesu kao naredbe pri prevođenju. Neke se koriste pri ispisu u početnoj funkciji (`OS_NAME`, `NAME_MAJOR`, `NAME_MINOR`, `ARCH` i `VERSION`), a neke i za druge potrebe (odabir arhitekture i imena izlaznih datoteka).

S lijeve strane znaka jednakosti (`=`, `:=`, `?=`, `+=`) mora biti ime varijable (bez znaka `$`). S desne strane može biti običan niz znakova (primjerice kao u `ARCH ?= i386`) ili varijabla, ali sada označena sa znakom `$` (primjerice kao u `PROJECT = $(NAME_MINOR)`). Varijabla može biti i polje ako s desne strane znaka jednakosti ima više vrijednosti odvojenih razmacima ili tabulatorima.

Za razliku od "normalnog" pridjeljivanja s `=` (izvorno *recursive*), pridjeljivanje s `:=` (izvorno *static*) traži trenutnu obradu (evaluaciju) svih varijabli s desne strane u trenutku čitanja (prolaska

kroz tu liniju).

Na primjer, ako bismo imali sljedeći niz postavljanja varijabli:

```
A = X
B = $(A)Y
C = $(B)Z
A = W
# ako se sada koriste varijable:
test:
        @echo A=$(A), B=$(B), C=$(C)
```

ispis će biti: `A=W, B=WY, C=WYZ`. Ako bi se umjesto = u gornjem linijama stavilo `:=`, tj. kada bi kôd bio:

```
A := X
B := $(A)Y
C := $(B)Z
A := W
# ako se sada koriste varijable:
test:
        @echo A=$(A), B=$(B), C=$(C)
```

ispis će biti: `A=W, B=XY, C=XYZ`.

U ovom je slučaju (korišteni `config.ini`) svejedno što se koristi (= ili `:=`) budući da se varijable s desne strane ne mijenjaju.

Osim navedenih pridruživanja, mogu se koristiti i dodavanje (izvorno *appending*) operatorom `+=` te uvjetna pridjela (izvorno *conditional*) operatorom `?=`. Operator dodavanja će proširiti varijablu s lijeve strane dodavanjem elemenata s desne strane operatora `+=`.

Uvjetni operator `?=` napravit će pridjelu vrijednosti varijabli tek ako varijabla još nije definirana (a može biti definirana u prethodnim linijama ili uključenoj datoteci ili među varijablama okoline ili u naredbenoj liniji). Primjerice, varijabla `ARCH` poprimit će vrijednost `i386` ako već prije nije poprimila vrijednost. Ako prevođenje pokrećemo naredbom:

```
$ make ARCH=arm
```

varijabla će imati vrijednost `arm`.

Varijable zadane u naredbenom retku "nadjačavaju" one zadane u datoteci (jer to ima smisla!) tako da u gornjem primjeru i nije bilo neophodno koristiti pridjeljivanje s `?=`. Međutim, ponekad bismo željeli obrnutu situaciju, da varijabla u datoteci nadjača onu iz naredbene linije. Za to trebamo korisititi ključnu riječ `override`. Primjerice, ako u datoteci stoji

```
override ARCH=i386
```

prijašnje pokretanje (`make ARCH=arm`) neće promijeniti tu varijablu.

Ostatak datoteke `config.ini` definira ostale potrebne varijable, kao što su ime direktorija gdje će se prevedeni kôd postavljati (`build`), početna adresa učitavanja programa u radni spremnik pri pokretanju, veličina stoga i druge. Većina varijabli je i kratko komentirana u kôdu.

Isječak kôda 5.8. Chapter_02_Source_tree/01_Source_tree/arch/i386/config.ini

```
15 # Intermediate and output files are placed into BUILDDIR
16 BUILDDIR = build
17
18
19 # Where will system be loaded when started (for which address to prepare it)
20 LOAD_ADDR = 0x100000
21
22 # System resources
```

```
23 #-----------------------------------------------------------------------------
24 STACK_SIZE = 0x1000
25
26 # System memory (in Bytes)
27 SYSTEM_MEMORY = 0x800000
28
29
30 # Library with utility functions (strings, lists, ...)
31 #-----------------------------------------------------------------------------
32 LIBS = lib
33
34
35 # Compiling and linking
36 #-----------------------------------------------------------------------------
37 LINK = ld
38 LDSCRIPT = $(BUILDDIR)/ARCH/boot/ldscript.ld
39 LDFLAGS = -melf_i386
40 LDFLAGS_OPT = -O3 --gc-sections -s
41 LDFLAGS_OPTD = -O3 --gc-sections
42
43 CC = gcc
44
45 CFLAGS = -m32 -march=i386 -Wall -Werror -nostdinc -ffreestanding -nostdlib -fno-
   stack-protector -fno-pie
46
47 # additional optimization flags
48 CFLAGS_OPT = -O3 -fdata-sections -ffunction-sections
49
50 #optimization with debug information
51 CFLAGS_OPTD = -O3 -fdata-sections -ffunction-sections -g
52
53 # Linker flags
54 #if in command line given: debug=yes or/and optimize=yes
55 ifeq ($(optimize),yes)
56 ifeq ($(debug),yes) #if both are set!
57 CFLAGS += $(CFLAGS_OPTD)
58 LDFLAGS += $(LDFLAGS_OPTD)
59 CMACROS += DEBUG
60 else
61 CFLAGS += $(CFLAGS_OPT)
62 LDFLAGS += $(LDFLAGS_OPT)
63 endif
64 else #debug set by default
65 CFLAGS += -g
66 CMACROS += DEBUG
67 endif
68
69
70 # directories to include while compiling
71 DIRS_K := arch/$(ARCH)/boot arch/$(ARCH) arch/$(ARCH)/drivers \
72         kernel $(LIBS)
73 DIRS_P := api programs/hello_world
74
75 DIRS := $(DIRS_K) $(DIRS_P)
76
77 # include dirs
78 INCLUDES := include $(BUILDDIR) include/api
79
80 QEMU_MEM = $(shell echo $$(( ($(SYSTEM_MEMORY)-1)/1048576+1 )) )
81 QEMU = qemu-system-$(ARCH)
82 QFLAGS = -m $(QEMU_MEM)M -machine accel=tcg
83 QMSG = "Starting qemu (pop-up window)"
```

Za prevođenje je ovdje definiran program *gcc*, a za povezivanje *ld*. I za povezivanje bi se teoretski mogao koristiti *gcc*, ali se za sada pokazao problematičnim (njegov povezivač zna

"zbuniti" GRUB koji ne pronalazi traženo zaglavlje i ne pokreće izgrađeni sustav, kao da ne koristi zadani `ldscript.ld`).

Zastavice za prevođenje (`CFLAGS`) i povezivanje (`LDFLAGS`) su vrlo slične već opisanima u datoteci `build.sh` u početnom inkrementu (tablica 4.1.).

Optimizacija kôda s obzirom na brzinu i veličinu vrlo je bitna za ugrađene sustave. Međutim, u postupku izrade programske potpore potrebna je podrška za praćenje izvođenja te ispravljanje grešaka (engl. *debugging*). Radi jednostavnosti promjene načina izgradnje uz korištenje optimiranja i/ili uz pripremu za praćenje rada, u `Makefile` je dodana kratka provjera ulaznih parametara. Pri pokretanju prevođenja Benua mogu se zadati `debug=yes` (praćenje rada) i `optimize=yes` (optimizacija kôda). Ako se ništa ne zada pretpostavit će se `debug=yes` (bez optimizacije).

Datoteke za prevođenje (`.c` i `.S`) tražit će se u direktorijima čija imena sadrži varijabla `DIRS`.

Datoteka `Makefile` nakon uključivanja prethodno opisane datoteke `config.ini` najprije definira još neke interno potrebne varijable.

Isječak kôda 5.9. Chapter_02_Source_tree/01_Source_tree/Makefile

```
6  #default target
7  ARCH ?= i386
8
9  CONFIG_INI = arch/$(ARCH)/config.ini
10 THIS_MAKEFILE = Makefile
11 CONFIG_FILES = $(CONFIG_INI) $(THIS_MAKEFILE)
12
13
14 include $(CONFIG_INI)
15
16 KERNEL_FILE_NAME = $(PROJECT).elf
17 BINFILE = $(BUILDDIR)/$(KERNEL_FILE_NAME)
18
19 CMACROS += OS_NAME="\"$(OS_NAME)\"" PROJECT="\"$(PROJECT)\""            \
20            NAME_MAJOR="\"$(NAME_MAJOR)\"" NAME_MINOR="\"$(NAME_MINOR)\""\
21            ARCH="\"$(ARCH)\"" AUTHOR="\"$(AUTHOR)\""            \
22            VERSION="\"$(VERSION)\""
23
24
25 #-----------------------------------------------------------------------------
26 # Misc
27
28 CMACROS += SYSTEM_MEMORY=$(SYSTEM_MEMORY)                                \
29         STACK_SIZE=$(STACK_SIZE)                                        \
30         LOAD_ADDR=$(LOAD_ADDR)
```

Radi povećanja prilagodljivosti pri podešavanju mnogo varijabli se podešava samo na jednom mjestu – datoteci `config.ini` te prenosi programima u obliku makroa. Varijabla `CMACROS` sadrži popis takvih varijabli u obliku `varijabla=vrijednost` ili samo u obliku postavljenih zastavica navođenjem imena zastavice `zastavica`. Svakoj takvoj varijabli ili zastavici će se u postupku prevođenja dodati prefix `-D` te s time predati programima *gcc* i *ld*.

Isječak kôda 5.10. Chapter_02_Source_tree/01_Source_tree/Makefile

```
34 all: $(BINFILE)
```

Prvi predmet naveden u datoteci zove se `all`. Prevođenje predmeta `all` aktivira se s `make all` ili samo s `make` s obzirom na to da je to prvi predmet i sukladno tome i pretpostavljeni za izradu. Nakon dvotočke (`:`) slijedi popis ovisnosti za ovaj predmet. Ovdje se on sastoji od samo jednog elementa: predmeta `$(BINFILE)`. Element `$(BINFILE)` je zapravo predmet koji za koji je kasnije definirano kako ga napraviti (zaseban predmet, a što je zapravo pravi cilj prevođenja). Predmet `all` je ovdje naveden samo zato da bude prvi.

Isječak kôda 5.11. Chapter_02_Source_tree/01_Source_tree/Makefile

```
37 FILES := $(foreach DIR,$(DIRS),$(wildcard $(DIR)/*.c $(DIR)/*.S))
38 OBJECTS := $(addprefix $(BUILDDIR)/,$(FILES:.c=.o))
39 OBJECTS := $(OBJECTS:.S=.asm.o)
40 DEPS := $(OBJECTS:.o=.d)
```

Gornje linije funkcijama `foreach`, `wildcard`, `addprefix` te implicitnim `$(VAR:.x=.y)` generiraju popise datoteka s izvornim kôdom koje treba prevesti (spremljeno u varijabli `FILES`), objekate koji će pri prevođenju nastati (imena datoteka navedena u `OBJS`) te datoteke s definicijom ovisnosti (`DEPS`) koje treba stvoriti. Funkcija `foreach` će za svaki zadani direktorij iz `DIRS` u varijablu `FILES` pribrojiti sve datoteke tog direktorija koje završavaju s `.c` i `.S` funkcijom `wildcard`. Popis objekata koje treba izgraditi dobiva se promjenom popisa datoteka u `FILES` uz zamjenu ekstenzije `.c` u `.o` te dodavanja prefiksa `$(BUILDDIR)` (vrijednosti te varijable). Ekstenziju za asemblerske izvorne datoteke u popisu objektnih treba zamijeniti s `.S` u `.asm.o`.

Primjerice, za direktorij `Chapter_02_Source_tree/01_Source_tree` varijabla `FILES` će imati vrijednosti (sve u istom retku, odvojene razmakom koji je ovdje označen znakom ␣):

```
arch/i386/boot/startup.S␣arch/i386/descriptor.c␣arch/i386/drivers/print.c␣kernel/
kprint.c␣kernel/startup.c␣lib/string.c␣api/stdio.c␣programs/hello_world/hello_world
.c.
```

Prije prevođenja potrebno je stvoriti direktorije u koje će se rezultati prevođenja staviti. Stvaranje tih direktorija obavlja predmet `$(DIRS_CREATED)`, a kao potvrdu da su direktoriji stvoreni stvara se jedna prazna datoteka `.null` koja će u idućim predmetima biti potvrda da su direktoriji stvoreni (oni će tu datoteku imati kao prvu u svojim ovisnostima).

Isječak kôda 5.12. Chapter_02_Source_tree/01_Source_tree/Makefile

```
42 # dummy file that indicate directories for objects are created
43 DIRS_CREATED = $(BUILDDIR)/.null
44
45 # create required directories in $(BUILDDIR) directory (including $(BUILDDIR))
46 # also create ARCH symbolic link for selected platform source
47 # (used for #include <ARCH/*> purposes)
48 $(DIRS_CREATED):
49         @-if [ ! -e $(BUILDDIR) ]; then mkdir -p $(BUILDDIR); fi;
50         @-$(foreach DIR,$(DIRS), if [ ! -e $(BUILDDIR)/$(DIR) ];        \
51                 then mkdir -p $(BUILDDIR)/$(DIR); fi; )
52         @ln -s ../arch/$(ARCH) $(BUILDDIR)/ARCH
53         @touch $(DIRS_CREATED)
```

Sklopovlje, tj. arhitektura za koju će se sustav izgrađivati odabire se varijablom `ARCH`. Da bismo iz definicije sučelja (`include/arch`) mogli povezati (uključiti) prava zaglavlja odabrane arhitekture korištenjem `#include <ARCH/nesto.h>` potrebno je napraviti simbolički link na direktorij s odabranom arhitekutrom (ili kopirati taj direktorij ako simbolički linkovi nisu podržani na korištenom datotečnom sustavu). Prethodni dio kôda radi i navedeno (linija 52).

U ovoj datoteci `Makefile` nije zadano kako se prevodi svaka izvorna datoteka zasebno, već se koriste "recepti", tj. pravila koja uključuju više datoteka.

Isječak kôda 5.13. Chapter_02_Source_tree/01_Source_tree/Makefile

```
55 # define how to compile .c files
56 $(BUILDDIR)/%.o: %.c $(CONFIG_FILES) $(DIRS_CREATED)
57         @echo [compiling] $< ...
58         @$(CC) -c $< -o $@ -MMD $(CFLAGS) \
59                 $(foreach INC,$(INCLUDES),-I $(INC)) \
60                 $(foreach MACRO,$(CMACROS),-D $(MACRO))
61
62 # define how to compile .S files (assembler)
```

```
63 $(BUILDDIR)/%.asm.o: %.S $(CONFIG_FILES) $(DIRS_CREATED)
64         @echo [compiling] $< ...
65         @$(CC) -c $< -o $@ -MMD $(CFLAGS) \
66                 $(foreach INC,$(INCLUDES),-I$(INC)) \
67                 $(foreach MACRO,$(CMACROS),-D $(MACRO))
```

Prvi gornji recept `$(BUILDDIR)/%.o` treba primijeniti za sve objekte koji započinju s imenom `$(BUILDDIR)`, a završavaju ekstenzijom .o. Ekvivalentno, drugi recept se odnosi na objekte koji završavaju s `.asm.o`.

S desne strane prve linije recepta (iza `:`), nalazi se `%.c` (`%.S`) što će povezati objekt s izvornom datotekom. Na primjer, objekt `build/kernel/kprint.o`, koji je definiran kao jedan od potrebnih objekata za izradu datoteke sa slikom sustava (prema popisu iz varijable `OBJS`), nastaje iz datoteke `kernel/kprint.c` s izvornim kôdom. Znak postotka `%` koji zamjenjuje proizvoljan niz znakova će u gornjem primjeru zamijeniti `kernel/kprint`. Ostala dva elementa ovisnosti su: datoteke s postavkama prevođenja (`Makefile` i `config.ini`) te datoteka koja kaže da su potrebni direktoriji za spremanje izlaza. Datoteke s postavkama određuju način prevođenja te svaka njihova promjena može promijeniti i način prevođenja pojedine datoteke. Zato se promjenom tih datoteka nanovo prevode i sve datoteke s izvornim kôdom.

Redak `@echo [compiling] $< ...` služi jedino radi ljepšeg i kraćeg ispisa pri prevođenju, s obzirom na to da se sama naredba `gcc` ne ispisuje (koja može biti i poprilično duga s obzirom na sve zastavice i makroe). Varijabla `$<` zamjenjuje se prvim elementom iza `:` iz prve linije predmeta, tj. imenom datoteke koja se prevodi (primjerice s `kernel/kprint.c`).

Zadnja naredba recepta poziva samo prevođenje. Varijabla `$@` zamjenjuje se imenom predmeta (primjerice `build/kernel/kprint.o`).

Zastavica `-MMD` kaže programu *gcc* da pored prevođenja i generiranja `.o` datoteke, također na istom mjestu napravi i datoteku ista imena uz sufiks `.d`, koja će sadržavati popis ovisnosti te datoteke (zaglavlja o kojima je datoteka ovisna). Ovisnosti su oblika:

```
datoteka.c: zaglavlje1.h zaglavlje2.h
```

Ta se datoteka pri idućem prevođenju uključuje u `Makefile` (pogledati njegovu zadnju liniju) te *make* može provjeriti jesu li se promijenila zaglavlja o kojima ovisi neka `.c` datoteka i ako jesu, datoteka se mora ponovno prevesti. Prvo će prevođenje i tako uključiti sve datoteke, s obzirom na to da njihovi "prijevodi" (`.o` datoteke) ne postoje. Kada se neka izvorna datoteka mijenja radi dodavanja zaglavlja (čija ovisnost nije prisutna u odgovarajućoj `.d` datoteci) datoteka će se ionako prevesti jer je mijenjana, a time će se i ažurirati ovisnosti.

Funkcijom `foreach` uključuju se svi makroi definirani varijablom `CMACRO` s prefiksom `-D` te svi direktoriji sa zaglavljima definirani u `INCLUDES` s prefiksom `-I`.

Isječak kôda 5.14. Chapter_02_Source_tree/01_Source_tree/Makefile

```
69 # preprocessed linker script (constants)
70 LDSCRIPT_PP := $(BUILDDIR)/ldscript.ld
71 $(LDSCRIPT_PP): $(LDSCRIPT) $(CONFIG_FILES) $(DIRS_CREATED)
72         @$(CC) -E -P -x c -o $@ $< $(CFLAGS)                    \
73                 $(foreach INC,$(INCLUDES),-I$(INC))             \
74                 $(foreach MACRO,$(CMACROS),-D $(MACRO))
```

Budući da su neke konstante zadane i za skriptu povezivača (engl. *linker script*), kao što je to početna adresa `LOAD_ADDR`, i tu skriptu je potrebno "obraditi" (engl. *preprocess*). Gornji recept koristi *gcc*, ali uz posebne zastavice: `-E` zaustavlja obradu nakon prvog prolaza, `-P` isključuje dodavanje oznake linije (u prvom prolazu), a `-x c` definira programski jezik C (bez obzira na ekstenziju ulazne datoteke). Takav osnovni prolaz datoteke će zamijeniti konstante (varijable) onima definiranim u datotekama `Makefile` i `config.ini` (obaviti fazu 'pretprocesiranja').

Isječak kôda 5.15. Chapter_02_Source_tree/01_Source_tree/Makefile

```
77 # OS image
78 $(BINFILE): $(OBJECTS) $(LDSCRIPT_PP)
79         @echo [linking $@]
80         @$(LINK) -o $@ $(OBJECTS) -T $(LDSCRIPT_PP) $(LDFLAGS)
```

Povezivanje objekata u sliku sustava definirano je predmetom `$(BINFILE)`. On je ovisan o svim objektima koji se ovdje povežu u jednu izlaznu datoteku imena `$(BINFILE)`.

Emulator QEMU se na raznim inačicama Linuxa pokreće na malo drukčije načine. U datoteku `config.ini` potrebno je navesti to ime. Idući predmet definira način pokretanja emulatora.

Isječak kôda 5.16. Chapter_02_Source_tree/01_Source_tree/Makefile

```
84 qemu: all
85         @echo $(QMSG)
86         @$(QEMU) $(QFLAGS) -kernel $(BINFILE)
```

Navedene zastavice definiraju svojstva i emulatora i emuliranog računala, a prethodno su definirane u `config.ini` (zadnje četiri linije). Primjerice, `-m 8` definira 8 MB spremnika za emulirano računalo, `-machine accel=tcg` isključuje korištenje ugrađenog modula u jezgri Linuxa (nije potreban), `-kernel $(BINFILE)` definira sliku sustava koja će se učitati te `-serial stdio` omogućuje da sve što se unese nakon pokretanja u naredbenom retku (na terminalu s kojeg je QEMU pokrenut) se prenosi prema emuliranom računalu korištenjem serijske veze tog računala i obratno, sve što dolazi preko serijske veze emuliranog računala prikazuje se na terminalu.

Isječak kôda 5.17. Chapter_02_Source_tree/01_Source_tree/Makefile

```
88 clean:
89         @echo Cleaning.
90         @-rm -f $(OBJECTS) $(DEPS) $(BINFILE)
91
92 clean_all cleanall:
93         @echo Removing build directory!
94         @-rm -rf $(BUILDDIR)
```

Predmeti `clean` te `cleanall` definiraju brisanje objekata, odnosno cijelog direktorija s objektima. Isti predmet može imati više imena, kao što su `cleanall` i `clean_all`. Svejedno što unesemo uz `make` – isti će se predmet aktivirati.

Isječak kôda 5.18. Chapter_02_Source_tree/01_Source_tree/Makefile

```
96 -include $(DEPS)
```

Ako ovo nije prvo prevođenje tada postoje već generirane ovisnosti (`.d` datoteke) te će ih gornja naredba uključiti u postupak prevođenja.

Znak minusa (–) prije neke naredbe kaže alatu *Make* da ne obustavlja svoj rad i u slučaju da mu naredba koja slijedi minus ne završi uspješno. Na primjer, ako ovisnosti ne postoje (za prvo prevođenje), oni se neće moći uključiti u `Makefile`, ali će *Make* svejedno nastaviti s radom (što inače ne bi napravio!).

U ovom su prikazu ukratko opisani neki mehanizmi koji se koriste u datoteci `Makefile`. Za detaljniji (i precizniji) opis rada alata *Make* (a time i elemente `Makefile` datoteke) pogledati opsežnije upute, kao što je [Make].

5.2. Datoteka `ldscript.ld`

U datoteci `i386/boot/ldscript.ld` nalaze se upute povezivaču (*linkeru*) kako posložiti izlaznu datoteku. Povezivanje ("linkanje") je postupak koji se obavlja nakon prevođenja ("kom-

pajliranja”) u kojem se od objektnih datoteka stvara konačna datoteka, u ovom slučaju “slika sustava” koja se učitava u ROM ugradbenog računala. Postupak prevođenja izvornih datoteka do stvaranja slike sustava ukratko je na primjeru prikazano tablicom 5.1.

Tablica 5.1. Ukratko: prevođenje, povezivanje, pokretanje

`dio1.c` (kod u C-u)	`dio2.c` (kod u C-u)	`dio3.S` (kod u asm.)	izvorni kod
`gcc -c dio1.c`	`gcc -c dio2.c`	`gcc -c dio3.S`	prevođenje
`dio1.o:` `.text, .rodata, .data, .bss`	`dio2.o:` `.text, .rodata, .data, .bss`	`dio3.o:` `.text, .rodata, .data, .bss`	objektne datoteke
`ld dio1.o dio2.o dio3.o -T ldscript.ld -o slika.elf`			povezivanje
`slika.elf:` `.text, .rodata, .data, .bss` (ili drugi prema `ldscript.ld`)			slika sustava
upisivanje slike u ROM (firmware) ugradbenog računala (ili pokretanje u simulatoru)			pokretanje

U početnom inkrementu je uputa povezivaču prenesena izravno kao argument naredbene linije, a sastoji se od početne adrese za koju se slika priprema (`-Ttext=0x100000`) te adrese na kojoj se nalazi prva instrukcija (`-e arch_start`). To u ostalim koracima nije dovoljno te se upute zadaju preko zasebne datoteke.

Svaka se prevedena datoteka – *objektna datoteka* (nastala prevođenjem datoteke s izvornim kôdom koja završava s `.c` ili `.S`) sastoji od *odjeljaka* (sekcija). Primjerice, pogledati ispis koji daje naredba `objdump -h ime_izlazne_datoteke` gdje izlazna datoteka može biti `.o` datoteka (nastala samo prevođenjem, bez povezivanje) ili program koji se može i pokrenuti (nastao prevođenjem i povezivanjem). Odjeljci mogu biti imenovani različitim imenima, ali prevoditelji uglavnom koriste ista imena. Uobičajeni nazivi odjeljaka su:

- `.text` za instrukcije,
- `.rodata` za konstante,
- `.data` za globalne varijable (sa zadanom početnom vrijednošću) te
- `.bss` za dinamičke podatke (stog, neinicijalizirane globalne varijable, dio spremnika koji se koristi u `malloc/free` funkcijama i slično).

Odjeljak `.bss` nije prisutan u izlaznoj datoteci (datoteci s programom, tj. slikom sustava), odnosno ne zauzima prostor već samo definira spremnički prostor koji treba stvoriti kada se program (sustav) učita u radni spremnik (i inicijalno popuniti nulama).

Kada se izlazna datoteka projekta treba posložiti od većeg broja datoteka, onda se povezivanje radi povezivanjem svih tih izlaznih datoteka (svih odjeljaka iz svih datoteka). Kada se radi o običnom programu, povezivač će sam posložiti odjeljke datoteka prema uobičajenim pravilima (sve instrukcije zajedno, konstante zajedno i slično). Međutim, kada se izgrađuje program ili operacijski sustav za ugrađeno računalo tada je često potrebno definirati gdje će se pojedini dio sustava nalaziti – na kojim adresama. Najjednostavniji primjer jest kada se instrukcije i konstante trebaju koristiti iz ROM-a, a sve ostalo treba najprije biti kopirano u RAM i korišteno iz njega. Opis kako posložiti izlaznu datoteku iz ulaznih datoteka opisuje se posebnom datotekom koja se koristi pri povezivanju. U ovom projektu ta se datoteka zove `ldscript.ld` (kasnije i `kernel.ld` i `user.ld`).

Glavni dio datoteke jesu definicije izlaznih odjeljaka, tj. koji se odjeljci kojih ulaznih datoteka slažu u koje odjeljke izlazne datoteke. Svaki odjeljak izlazne datoteke definira što ide u njega, ali i gdje treba učitati taj odjeljak pri pokretanju te za koju adresu se odjeljak priprema (ne mora biti ista kao i adresa na koju će se početno učitati). Ukratko bismo odjeljak mogli opisati sljedećim formatom:

```
    /* unutar SECTIONS dijela */
    ime_odjeljka [adresa za koju se priprema] : [AT(adresa na koju će se učitati)]
    {
         datoteke_1 ( odjeljci_1 )
        [datoteke_2 ( odjeljci_2 )]
        [...]
    }
```

Primjerice, ako se radi o varijablama koje će se početno nalaziti u ROM-u, ali pri pokretanju će se prekopirati u RAM i nadalje koristiti iz RAM-a, tada će *adresa na koju će se učitati* biti adresa u ROM-u, a *adresa za koju se priprema* u RAM-u. Dio `datoteke_1` može biti popis datoteka ili filtar koji može uključivati više datoteka (npr. `*.o`). Slično, `odjeljci_1` može biti popis imena odjeljaka koji će se iz tih datoteka uzimati ili filtar koji uključuje više njih (npr. `*.*data*`). Primjerice, ako želimo varijable (koje se nalaze u odjeljku `.data`) iz svih datoteka u direktoriju `prvi` staviti zajedno u izlazni odjeljak `.var1` koji se treba učitati u ROM na adresu 0x1000000, ali kasnije koristiti iz RAM-a na adresi 0x2000000 onda bi definicija tog odjeljka bila `.var1 0x2000000:AT(0x1000000){ prvi/?*.o (.data)}`.

Na ovom je mjestu datoteka `ldscript.ld` samo ukratko opisana. Nešto detaljniji opis nalazi se uz inkrement kojim se objašnjavaju mogućnosti upravljanja spremničkim prostorom (odjeljci 12.4. i 12.5.2.), a još složeniji primjeri nalaze se u dodatku D.

Isječak kôda 5.19. Chapter_02_Source_tree/01_Source_tree/arch/i386/boot/ldscript.ld

```
5  ENTRY(arch_startup)
```

Gornja oznaka definira adresu gdje se nalazi početak kôda (adresa prve instrukcije programa, tj. operacijskog sustava). U ovom sustavu početna je funkcija definirana u `startup.S` datoteci.

Glavni dio datoteke su definicije odjeljaka.

Isječak kôda 5.20. Chapter_02_Source_tree/01_Source_tree/arch/i386/boot/ldscript.ld

```
7  SECTIONS {
8          .code LOAD_ADDR :
9          {
10                 kernel_code_addr = .;
11
12                 /* instructions */
13                 *?/boot/startup.asm.o ( .text* )
14
15                 *( .text* )
16         }
17         .data :
18         {
```

```
19                  kernel_data_addr = .;
20
21                  /* read only data (constants), initialized global variables */
22                  * ( .rodata* .data* )
23          }
24          .bss :
25          {
26                  *( .bss* COMMON* )
27
28                  . = ALIGN (4096);
29          }
30  #ifndef DEBUG
31          /DISCARD/ : { *(*) }
32  #endif
33          kernel_end_addr = .;
34  }
```

Oznaka pojedinog izlaznog odjeljka započinje s imenom, opcionalnom adresom za koju pripremiti odjeljak (konstanta `LOAD_ADDR` definirana u `config.ini`), dvotočkom (:) te adresom kamo odsječak smjestiti pri učitavanju (na koju adresu). Ako zadnja adresa nije zadana uzima se prethodna adresa (smještanje na adresu za koju se i priprema). Navedena datoteka definira izlazne odjeljke naziva `.code`, `.data` i `.bss`. Imena izlaznih odjeljaka za ovaj sustav nisu bitna s obzirom na to da se sustav samo jednom učitava (ne koristi uobičajene alate za naknadano učitavanje i prepoznavanje odjeljaka).

Za svaki izlazni odjeljak treba definirati koji elementi ulaze u njega, tj. koji dijelovi prevedenih datoteka (dijelovi `.o` datoteka) će se u njega smjestiti. Zbog toga su u ovoj datoteci (`ldscript.ld`), za svaki izlazni odjeljak definirani ulazni odjeljci i ostali elementi koji će se u izlaznome odjeljku pojaviti pri povezivanju.

Kako *multiboot* zaglavlje mora biti smješteno pri početku datoteke, u izlaznoj se datoteci najprije pojavljuje datoteka sa zaglavljem, tj. `*?/boot/startup.asm.o`. Oznaka `*?` zamjenjuje bilo koji niz znakova, a zapravo se radi o imenu direktorija u koji će se smjestiti izlazne datoteke (`*` zamjenjuje proizvoljan niz znakova, a `?` zamjenjuje samo jedan znak i dodan je da se `*/` ne interpretira kao oznaka kraja komentara). U ovom je slučaju zapis mogao biti i kraći, primjerice `*?/startup.asm.o` s obzirom na to da postoji samo jedna `startup.S` datoteka.

Nakon filtra za datoteke (što `*?/boot/startup.asm.o` jest) dolazi popis odjeljaka koje će se iz tih datoteka prenijeti u izlaznu sekciju. Taj popis u prvom slučaju uključuje sve odjeljke čija imena započinju s `.text`.

Nakon `.text` odjeljka `startup.asm.o` datoteke, u izlazni se odjeljak stavljaju svi `.text` odjeljci svih ostalih datoteka prema idućoj naredbi: `*( .text* )`.

Izlazni odjeljak `.data` sadrži podatke (varijable i konstante), tj. odjeljke `.rodata*` i `.data*`.

Konačno, izlazni odjeljak `.bss` sadrži odjeljke `.bss*` i `COMMON*`.

Naredba `. = ALIGN (4096);` definira da kraj odjeljka `.bss` treba poravnati (proširiti) prema navedenom broju (prva iduća adresa djeljiva je s `4096`). Trenutno ovo i nije potrebno, s obzirom na to da je na izlazu samo jedan odjeljak. Međutim, to postaje potrebno u idućim koracima kada bude više izlaznih odjeljaka (primjerice GRUB neće učitati sliku bez takva poravnanja).

Naredba `/DISCARD/ : { *(*)}` nalaže da se svi ostali odjeljci koji nisu navedeni do sada iz svih datoteka zanemare (ne uključuju u konačnu datoteku). Tu spadaju posebni odjeljci koje prevoditelj stvara, a koji se koriste u razne svrhe (primjerice pri 'debugiranju'). Za normalan rad sustava ti odjeljci nisu potrebni pa se odbacuju (kada se ne koristi ispitni 'DEBUG' način rada).

U skripti se još nalaze varijable `kernel_code_addr`, `kernel_data_addr` te `kernel_end_addr` kojima je pridijeljena vrijednost lokacije gdje se nalaze (`varijabla = .;`). Navedene

varijable mogu poslužiti za određivanje adrese. Primjerice, u nekoj C datoteci mogli bismo se pozvati na te varijable da dođemo do njihovih vrijednosti.

```
    extern char kernel_end_addr;
    ...
    adresa_kraja = (void *) &kernel_end_addr;
```

Za dohvat vrijednosti se koriste adrese, a ne vrijednosti koje smo varijablama pridijelili u skripti [Binutils2]. Razlog je što to zapravo nisu varijable u pravom smislu, kao što bi to bile varijable u C-u. Naime, za varijable u C-u stvara se "simbol" u tablici simbola koji sadrži adresu lokacije u memoriji gdje se pohranjuje vrijednost varijable. Međutim, simbol nastao na osnovu skripte za povezivanje nije povezan s vrijednošću u memoriji. Zato se njegova adresa koristi kao vrijednost.

Kako izgleda raspored odjeljaka pojedinog programa ili objektne datoteke može se vidjeti naredbom `objdump` (pokrenutom nakon prevođenja sustava – nakon pokretanja naredbe `make`). Na primjer, s `objdump -h build/01_Source_tree.elf` vide se odjeljci u slici sustava.

```
$ objdump -h build/01_Source_tree.elf

build/01_Source_tree.elf:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .code         00000425  00100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         0000008f  00100440  00100440  00001440  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00001b20  001004e0  001004e0  000014cf  2**5
                  ALLOC
$
```

Pri uobičajenom prevođenju pojavljuje se više izlaznih odjeljaka. Na primjer, ako se pogleda datoteka `build/arch/i386/drivers/print.o` to se jasno vidi.

```
$ objdump -h build/arch/i386/drivers/print.o

build/arch/i386/drivers/print.o:     file format elf32-i386

Sections:
Idx Name     Size     VMA      LMA      File off Algn
 0 .text     00000000 00000000 00000000 00000034 2**2
             CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data     00000000 00000000 00000000 00000034 2**2
             CONTENTS, ALLOC, LOAD, DATA
 2 .bss      00000000 00000000 00000000 00000034 2**2
             ALLOC
 3 .text.arch_console_init 00000025 00000000 00000000 00000040 2**4
             CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 4 .text.arch_console_print_word 000000ec 00000000 00000000 00000070
             2**4       CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 5 .bss.row.865 00000004 00000000 00000000 0000015c 2**2
             ALLOC
 6 .comment   0000002b 00000000 00000000 0000015c 2**0
             CONTENTS, READONLY
 7 .note.GNU-stack 00000000 00000000 00000000 00000187 2**0
             CONTENTS, READONLY
 8 .eh_frame   0000007c 00000000 00000000 00000188 2**2
             CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
$
```

U navedenom je ispisu izvorni `.text` odjeljak prazan, a sav kôd je u odjeljcima `.text.arch_console_init` i `.text.arch_console_print_word`. Razlog tomu je u korištenju zastavice `-ffunction-sections` za *gcc* koja nalaže da svaka funkcija bude u

svom odjeljku. Navedena zastavica omogućava da se pri povezivanju oni odjeljci koji se ne koriste i ne uključuju u izlaznu datoteku (zastavica `--gc-sections` za *ld*).

Izgled, broj i imena odjeljaka ovise o prevoditelju (i njegovoj inačici) koji se koristi (a koji je i ovisan o sustavu na kojem se nalazi).

Ukratko, `ldscript.ld` definira način izgradnje sustava iz objektnih datoteka, preslikavanje iz mnoštva `*.o` datoteka u jednu `.elf` datoteku (kada je izlazni format ELF).

5.2.1. Dodatni primjer korištenja skripte za povezivanje

U direktoriju `Chapter_02_Source_tree/04x_ldscript_example` nalazi se nešto složeniji primjer korištenja skripte za povezivanje koji je dijelom prikazan u nastavku. U primjeru je samo dio koda pripremljen za izvođenje iz "ROM"-a dok sve ostalo treba prvo premjestiti u RAM, instrukcije jezgre na adresu 0x50000, konstante i podatke jezgre na 0x60000, instrukcije programa na 0x70000, podatke programa na 0x80000 te stog postaviti na adresu 0x90000.

Isječak kôda 5.21. Chapter_02_Source_tree/04x_ldscript_example/arch/i386/boot/ldscript.ld

```
/*! linker script for memory layout */

ENTRY(arch_startup)

ROM_ADDR      = 0x10000;
RAM_ADDR      = 0x50000;
KERN_CODE_RAM = 0x50000;
KERN_DATA_RAM = 0x60000;
PROG_CODE_RAM = 0x70000;
PROG_DATA_RAM = 0x80000;
STACK_ADDR    = 0x90000;

SECTIONS {
        boot_code_ROM = ROM_ADDR;
        .boot_code boot_code_ROM : AT (boot_code_ROM)
        {
                /* runs from ROM */
                build/arch/?*?/boot/?* ( .text* .rodata* .data* .bss*)
        }

        kern_code_ROM = boot_code_ROM + SIZEOF(.boot_code);
        .kern_code KERN_CODE_RAM : AT (kern_code_ROM)
        {
                /* copy to RAM and then run from there */
                build/arch/?* ( .text* )
                build/kernel/?* ( .text* )
                build/lib/?* ( .text* )

                /* or: build/arch/?* build/kernel/?* build/lib/?* (.text*) */
                /* or: *(EXCLUDE_FILE(build/api/?* build/programs/?*) .text*) */
        }
        kern_code_size = SIZEOF(.kern_code);

        kern_data_ROM = kern_code_ROM + kern_code_size;
        .kern_data KERN_DATA_RAM : AT (kern_data_ROM)
        {
                /* copy to RAM and then use from there */
                build/arch/?* ( .rodata* .data* .bss* COMMON* )
                build/kernel/?* ( .rodata* .data* .bss* COMMON*  )
                build/lib/?* ( .rodata* .data* .bss* COMMON*  )
        }
        kern_data_size = SIZEOF(.kern_data);

        prog_code_ROM = kern_data_ROM + kern_data_size;
        prog_code PROG_CODE_RAM : AT (prog_code_ROM)
```

```
46          {
47                  build/api/?* ( .text* )
48                  build/programs/?* ( .text* )
49          }
50          prog_code_size = SIZEOF(prog_code);
51
52          prog_data_ROM = prog_code_ROM + prog_code_size;
53          prog_data PROG_DATA_RAM : AT (prog_data_ROM)
54          {
55                  build/api/?* ( .rodata* .data* .bss* COMMON* )
56                  build/programs/?* ( .rodata* .data* .bss* COMMON* )
57          }
58          prog_data_size = SIZEOF(prog_data);
59
60          kernel_end_addr = PROG_DATA_RAM + prog_data_size;
61
62          /* For stack, two options are shown here */
63          /* 1. treat system_stack as a variable in special section and
64                mark it as such in kernel/startup.c */
65          .stack_segment STACK_ADDR : AT (prog_data_ROM + prog_data_size)
66          {
67                  * ( .stack_section* )
68          }
69
70          /* 2. define system_stack here instead in kernel/startup.c */
71          /* system_stack = STACK_ADDR; */
72
73          /DISCARD/ : { *(* .*) }
74  }
```

Promjene u ostatku koda uključuju kod u datoteci `copy_to_RAM.c` iz direktorija `boot`, poziv istoimene funkcije iz te datoteke iz početna koda u datoteci `boot/startup.S` te oznaku uz varijablu kojom se definira stog u datoteci `kernel/startup.c`. Ispis adresa, radi prikaza ostvarenja navedena preseljenja, ostvaren je dodatnim kodom u `programs/debug/debug.c` Navedeni sadržaji prikazani su u nastavku.

Isječak kôda 5.22. Chapter_02_Source_tree/04x_ldscript_example/arch/i386/boot/startup.S

```
36          call copy_to_RAM
```

Isječak kôda 5.23. Chapter_02_Source_tree/04x_ldscript_example/arch/i386/boot/copy_to_RAM.c

```
5   void just_copy(char *from, char *to, size_t size)
6   {
7           size_t i;
8           for (i = 0; i < size; i++)
9                   to[i] = from[i];
10  }
11
12  void copy_to_RAM()
13  {
14          extern char kern_code_ROM, KERN_CODE_RAM, kern_code_size;
15          extern char kern_data_ROM, KERN_DATA_RAM, kern_data_size;
16          extern char prog_code_ROM, PROG_CODE_RAM, prog_code_size;
17          extern char prog_data_ROM, PROG_DATA_RAM, prog_data_size;
18
19          just_copy(&kern_code_ROM, &KERN_CODE_RAM, (size_t) &kern_code_size);
20          just_copy(&kern_data_ROM, &KERN_DATA_RAM, (size_t) &kern_data_size);
21          just_copy(&prog_code_ROM, &PROG_CODE_RAM, (size_t) &prog_code_size);
22          just_copy(&prog_data_ROM, &PROG_DATA_RAM, (size_t) &prog_data_size);
23  }
```

Isječak kôda 5.24. Chapter_02_Source_tree/04x_ldscript_example/kernel/startup.c

```
/*! kernel stack */
uint8 system_stack [ STACK_SIZE ] __attribute__((section(".stack_section")));
```

Isječak kôda 5.25. Chapter_02_Source_tree/04x_ldscript_example/programs/debug/debug.c

```
        void k_startup();
        extern char system_info[];

        printf("\n");
        LOG(INFO, " k_startup (kernel code) is at %x", k_startup);
        LOG(INFO, " startup (kernel data) is at   %x", system_info);
        LOG(INFO, " debug (program code) is at    %x", debug);
        LOG(INFO, " x (program data) is at        %x", &x);
        LOG(INFO, " a (variable on stack) is at   %x", &a);
```

5.3. Ispis znakova na zaslon

U početnoj fazi se koristilo sučelje za ispis niza znakova u prvu liniju zaslona. To je u idućoj fazi (`02_Console`) prošireno s nešto više mogućnosti ispisa.

U `include/types/io.h` definirano je sučelje za ispis na konzolu `console_t`.

Isječak kôda 5.26. Chapter_02_Source_tree/02_Console/include/types/io.h

```
/*! Console interface */
typedef struct _console_t_
{
        int (*init)(int flags);
        int (*print)(char *text);
}
console_t;
```

Sučelje definira operacije za brisanje zaslona, pomicanje značke (engl. *cursor*) te ispis niza znakova počevši od trenutnog mjesta značke.

Za sada jedino ostvarenje tog sučelja ostvareno je u `arch/i386/drivers/vga_text.c`. Promjena naprave za ispis na neku drugu napravu mogla bi se za tu napravu obaviti tako da se ostvari upravljački program korištenjem `console_t` sučelja.

Sučelje za ispis koje `console_t` nudi preko poziva `print` ispisuje niz znakova na zaslon. Uobičajena funkcija ispisa u C-u je `printf` (slična postoji i u drugim programskim jezicima) koja omogućava *formatirani ispis*. U okviru pomoćnih funkcija, u datoteci `lib/string.c`, ostvarena je funkcija formatiranog ispisa (samo osnovni formati) u niz znakova:

Isječak kôda 5.27. Chapter_02_Source_tree/02_Console/include/lib/string.h

```
int vssprintf(char *str, size_t size, char **arg);
```

koja omogućava ostvarenje funkcija `kprintf` (iz jezgre) i `printf` (iz sloja *api*), a koje su slične funkciji `printf` (ostvaruju ispis za samo podskup ulaznih podataka).

Sučelje `printf` podržava nekoliko upravljačkih kodova (engl. *ANSI escape sequence*) [ANSI code] koji se mogu koristiti za promjenu boje ispisanog teksta, pomicanje značke te brisanje zaslona. Podržani kodovi navedeni su u `vga_text.c` datoteci, uz primjere korištenja u `hello_world.c` i `kprint.c`.

5.3.1. O korištenje struktura preko skraćenog naziva

U Benu su gotovo sva sučelja kao i opisnici raznih elemenata sustava strukture (definirane sa `struct` u C-u). Svaka od tih struktura se koristi preko skraćena tipa definiranog preko ključne

riječi `typedef`.

U praksi se često koristi načelo da se u kodu koji koristi elemente strukture koristi puni naziv strukture a ne skraćeni. Primjerice, ako postoji struktura `struct console` onda se u kodu koji pristupa elementima te strukture (`init`, `print`) varijabla označava tipom `struct console`. U ostatku koda (drugim datotekama) koje ne koriste njene elemente, tj. čak je poželjno da niti "ne znaju" koji su, poželjno je koristiti skraćeni tip `console_t`, koje je definirano sa: `typedef struct console console_t`. U Benu nije (još) korišteno navedeno načelo, već su sve strukture dobilo skraćeno ime i preko njega se koriste.

5.4. Korištenje ulazno-izlaznih naprava

U `02_Console` su ponešto promijenjene i zastavice *multiboot* zaglavlja te se pri pokretanju (`arch/i386/boot/startup.S`) pohranjuju parametri koji se predaju sustavu (koji se u idućim inkrementima koriste pri inicijalizaciji spremničkog prostora te učitavanju zasebnih programa kao modula).

Datoteka `arch/i386/io.h` sadrži nekoliko funkcija za slanje i primanje i podataka i naredbi prema vanjskim jedinicama. Arhitektura x86 neke vanjske jedinice preslikava u spremnički prostor te im se tako može izravno pristupiti, dok je za ostale jedinice potrebno koristiti instrukcije `in` i `out`. Upravo ovo drugo načelo koriste funkcije u datoteci `arch/i386/io.h`, tj. olakšavaju komunikaciju s takvim jedinicama definiranjem sučelja u jeziku C. Slijedi ispis i objašnjenje jedne od njih.

Isječak kôda 5.28. Chapter_02_Source_tree/02_Console/arch/i386/io.h

```
9  /*!
10  * Write to 8-bit port
11  * \param port  Port number
12  * \param data  Data to be sent
13  */
14 static inline void outb(uint16 port, uint8 data)
15 {
16         asm ("outb %b0, %w1" : : "a" (data), "d" (port));
17 }
```

Ključne riječi `static inline` definiraju da se ove funkcije ne pozivaju kao obične funkcije, već da se pri prevođenju ugrade na mjesto od kuda se pozivaju (`inline`). Krajnje pojednostavljeno, ako bi u kôdu datoteke koja uključuje ovo zaglavlje stajalo:

```
outb ( 0x10, 0x20 );
```

tada bi prevoditelj tu liniju najprije trebao zamijeniti linijom:

```
    asm ( "outb %b0, %w1" : : "a" (0x20), "d" (0x10) );
```

prije generiranja kôda.

Ako bi prevoditelj ignorirao `inline` naredbu (`inline` je zapravo samo preporuka prevoditelju), a zaglavlje se uključi u više datoteka, nastao bi problem jer bismo istu funkciju imali u više datoteka. Te se datoteke međusobno ne bi mogle povezati, jer bi imale istu globalnu oznaku – oznaku funkcije na više mjesta te bi povezivač javio grešku. Zato uz `inline` ide i `static` koji kaže da funkcija nije globalna već samo dio datoteke u kojoj je definirana (simbol koji se generira nije globalan).

Jedina linija kôda prikazane funkcije (linija 16) je asemblerska naredba (engl. *inline assembly*). Prevoditelj bi tu naredbu trebao proslijediti generatoru kôda (uz eventualno dodatne instrukcije). Konkretno, ovdje se generira instrukcija `outb` s parametrima `data` i `port`. Međutim, s obzirom na to da takva instrukcija koja prima dva broja ne postoji, prevoditelj će gornju naredbu morati zamijeniti s bar tri instrukcije. Na primjer, u prvoj će u registar `al` postaviti

podatak `data`. Oznaka `"a"` prije (`data`) definira odredišni registar `al` za `data`, a `%b0` uz `outb` kaže da se radi o oktetu – `b` i prvom operandu – `0`). U drugoj instrukciji će u registar `dx` staviti broj izlaza (engl. *port*). Oznaka `"d"` prije `(port)` specificira registar `dx`, a `%w1` u instrukciji kaže da se radi o drugom parametru koji je duljine riječi – `w` od *word*, tj. 16 bita. Tek će se u trećoj instrukciji pokrenuti slanje na vanjsku jedinicu instrukcijom `outb %al, %dx`. Ako se u zadanim registrima već nešto nalazi to će trebati prije i pohraniti, a kasnije i obnoviti (primjerice sa stoga). Sve navedene dodatne instrukcije generira prevoditelj (*gcc*).

Korištenjem naredbe `asm` zajedno s proširenim sučeljem koje *gcc* omogućava značajno se olakšava pisanje kôda koji ponekad zahtjeva pokretanje određenih instrukcija procesora. Ovakve naredbe u ovom projektu javljaju se samo u sloju *arch* gdje i pripadaju.

5.5. Pomoć pri traženju grešaka

Programiranje neizbježno podrazumijeva i unošenje pogrešaka. Njihovo otkrivanje i otklanjanje može biti vrlo zahtjevno. Jedan od načina otkrivanja grešaka je u ubacivanju dodatnih ispisa i provjera. Međutim, i nakon što je greška pronađena i ispravljena, nije preporučeno micati dodani kôd za ispis i provjere, jer možda greška i nije u potpunosti odstranjena. Uobičajeno načelo u takvim okolnostima je u korištenju posebnih funkcija i makroa koji će za vrijeme probnog rada ispisivati sve ugrađene poruke i obavljati opsežnija ispitivanja uvjeta. Ta se dodatna ispitivanja i ispisi mogu isključiti jednostavnim makroom, tj. definiranjem ili nedefiniranjem određene varijable prije izgradnje sustava. U razmatranom projektu od faze `03_DEBUG` koristi se makro `DEBUG` u tu svrhu.

Funkcije (makroi) za ispis i ispitivanje su:

- `LOG(LEVEL, format, ...)`
- `ASSERT(expr)`
- `ASSERT_ERRNO_AND*`.

Primjeri korištenja navedenih funkcija prisutni su u cijelom kôdu u svim daljnjim koracima.

Pisanje makroa zahtjeva osnovno poznavanje sintakse te njegovih mogućnosti radi učinkovitijeg i kraćeg kôda. Primjerice, makro `LOG` koristi nekoliko naprednijih mogućnosti.

Isječak kôda 5.29. Chapter_02_Source_tree/03_DEBUG/include/kernel/errno.h

```
11 #define LOG(LEVEL, format, ...) \
12 kprintf("[" #LEVEL ":%s:%d]" format "\n", __FILE__, __LINE__, ##__VA_ARGS__)
```

Kada makro sadrži parametre treba paziti da početna zagrada i prvi parametar ne budu odvojeni dodatnim razmacima! Tri točke (`...`) kao zadnji parametri makroa označavaju da on prima varijabilan broj parametara – osim `LEVEL` i `format`, makro može primiti još parametara iza njih. Znak `\` na kraju linije označava da se linija zapravo nastavlja tekstom iz sljedeće linije (prevoditelj tako zapravo iduću liniju stavlja na kraj prethodne). Znak `#` ispred `LEVEL` označava operator pretvaranja imena koje slijedi iza znaka `#` u niz znakova (engl. *stringification*). Konstante `__FILE__` i `__LINE__` pri prevođenju će se zamijeniti imenom datoteke i linijom u kojoj se navedeni makro koristi (poziva).

Ako se makrou prenesu dodatni parametri, oni će se nadodati na kraju funkcije `kprintf` pomoću `##__VA_ARGS__`. Znakovi `##` predstavljaju operator spajanja koji spaja imena lijevo i desno od njega. U gornjem slučaju će se svi dodatni parametri označeni s `__VA_ARGS__` nadodati na kraj funkcije. Za primjer korištenja operatora `##`:

```
#define INC(X) (X) ## .cnt++
```

poziv `INC(packets);` generirao bi kôd `(packets).cnt++;`.

Primjerice, kada se gornji makro `LOG` pozove preko:

```
LOG ( ERROR, "Indeksi van granica: i1=%d, i2=%d", i1, i2);
```

tada će se on u početnoj fazi prevođenja (preprocesoru) prevesti u:

```
kprintf ( "[ERROR:%s:%d]Indeksi van granica: i1=%d, i2=%d\n",
          "program/test/test.c", 42, i1, i2);
```

te u slučaju poziva s parametrima `i1 = 1` i `i2 = 2` ispisati:

```
[ERROR:program/test/test.c:42]Indeksi van granica: i1=1, i2=2
```

Drugi način otkrivanja grešaka je postupak ispitnog pokretanja (engl. *debugging*). Takvo se otkrivanje uobičajeno rabi za obične aplikacije i vrlo je efikasno u otkrivanju grešaka s obzirom na mogućnosti zaustavljanja programa u svakom trenutku. Međutim, takva ispitivanja za izgrađeni operacijski sustav moguća su jedino uz prikladne simulatore i alate. Jedni on njih su kombinacija QEMU – GDB (engl. *The GNU project debugger*).

Emulator QEMU omogućava spajanje s GDB-om (na razne načine) i ispitivanje sustava kao da je običan program. U nastavku je kratko opisan jedan način pokretanja sustava korištenjem navedenih alata u svrhu otkrivanja grešaka.

Alat GDB za svoj rad treba sliku sustava koja sadržava ubačene simbole prikladne za ispitivanja. To zapravo znači da se pri prevođenju s alatom *gcc* treba uključiti zastavica `-g`. Također je poželjno isključiti optimizacije i kod prevođenja (`gcc`) i kod povezivanja (`ld`). Inače se neke varijable neće pojaviti u konačnoj inačici kôda i neće ih se na jednostavan način moći provjeravati (primjerice pri optimizaciji te su varijable u registrima procesora).

Ispitivanje je najbolje raditi iz dvije konzole: iz jedne će se pokrenuti `qemu`, a iz druge `gdb`.

```
[u prvoj konzoli]
$ qemu-system-i386 -m 8 -s -S -machine accel=tcg -kernel <slika.elf>

[u drugoj konzoli]
$ gdb -ex 'target remote localhost:1234' -s <slika.elf>
```

Zamijeniti `<slika.elf>` imenom stvarne slike sustava (npr. `build/04_Debugging.elf`).

Navedeno se može i jednostavnije pokrenuti korištenjem već pripremljenih recepata u datoteci `Makefile`: `make debug_qemu` te `make debug_gdb` u drugoj konzoli, a koji su dodani u `Makefile` od faze `04_Debugging`.

Isječak kôda 5.30. Chapter_02_Source_tree/04_Debugging/Makefile

```
95  # DEBUGGING
96  # For debugging to work: include '-g' in CFLAGS and omit -s and -S from LDFLAGS
97  # Best if -O3 flag is also omitted from CFLAGS and LDFLAGS (or some variables
98  # may be optimized away)
99  # Start debugging from two consoles:
100 #       1st: make debug_qemu
101 #       2nd: make debug_gdb
102 debug_qemu: all
103         @echo $(QMSG)
104         @$(QEMU) $(QFLAGS) -kernel $(BINFILE) -s -S
105 debug_gdb: all
106         @echo Starting gdb ...
107         @$(DEBUG_GDB) -s $(BINFILE) -ex 'target remote localhost:1234'
```

Rad s GDB-om prikazan je u nastavku na kratkom primjeru navedenom u datoteci `programs/debug/debug.c`:

```
1  (gdb) list programs/debug/debug.c:20
2  15      int debug ()
```

```
16      {
17              int a, b, c;
18
19              printf ( "Example program: [%s:%s]\n%s\n\n", __FILE__,
20                       __FUNCTION__, debug_PROG_HELP );
21
22              a = 1;
23
24              b = a + 1;
(gdb)
25
26              c = inc ( a ) + inc ( b );
27
28              a += b + c;
29              b += a + c;
30              c += a + b;
31
32              printf ( "a=%d, b=%d, c=%d\n", a, b, c );
33
(gdb)  break programs/debug/debug.c:24
Breakpoint 1 at 0x1012d6: file programs/debug/debug.c, line 24.
(gdb) c
Continuing.

Breakpoint 1, debug () at programs/debug/debug.c:24
24              b = a + 1;
(gdb) p a
$1 = 1
(gdb) n
26              c = inc ( a ) + inc ( b );
(gdb) s
inc (n=1) at programs/debug/debug.c:10
10              n++;
(gdb) p n
$2 = 1
(gdb) finish
Run till exit from #0  inc (n=1) at programs/debug/debug.c:10
0x001012ea in debug () at programs/debug/debug.c:26
26              c = inc ( a ) + inc ( b );
Value returned is $3 = 2
(gdb) n
28              a += b + c;
(gdb)
29              b += a + c;
(gdb)
30              c += a + b;
(gdb)
32              printf ( "a=%d, b=%d, c=%d\n", a, b, c );
(gdb)
```

Nekoliko osnovnih naredbi za rad s GDB-om:

- `break (b)` – postavljanje prekidne točke u kojoj će program stati i omogućiti ispitivanja; ponekad se može koristiti slična naredba `hbreak`;
- `continue (c)` – nastavak rada do iduće prekidne točke ili drugog događaja;
- `next (n)` – izvođenje iduće linije kôda (te ispis one iza);
- `step (s)` – izvođenje idućeg dijela kôda – pretpostavlja i ulazak u funkcije koje se pozivaju, ako su one u idućoj liniji;
- `finish` – izvođenje do kraja tekuće funkcije i izlazak iz nje;
- `print (p)` – ispis varijable ili nečeg drugog;

- `list (l)` – ispis nekoliko linija kôda u okolici trenutne pozicije ili ispis nekoliko linija kôda zadane datoteke;
- `backtrace (bt)` – ispis okvira stoga.

Neke naredbe imaju i kraći zapis koji je u gornjem popisu stavljen u zagrade.

Pritiskom na [enter] tipku bez zadavanja naredbe ponavlja se zadnja naredba.

Pitanja za vježbu 5

1. Zašto se često koristi načelo *podijeli i vladaj*?
2. Navesti podsustave operacijskog sustava.
3. Navesti slojeve u kojima se operacijski sustav izgrađuje.
4. Slojevita izgradnja operacijskih sustava (slojevi *arch/kernel/api/programs*) ima svoje prednosti i nedostatke. Navedite neke prednosti i nedostatke.
5. Opisati sadržaj datoteke s uputama za povezivača.
6. Koji se odjeljci stvaraju pri prevođenju C datoteke? Koji dijelovi datoteke idu u koje odjeljke?
7. Čemu služi `Makefile`? Opisati sadržaj te datoteke u Benu sustavu.
8. Opisati način ostvarenja makroa `LOG`. Kako se koriste navedeni elementi makroa (i u drugim primjerima)?
9. Korištenjem kombinacije alata QEMU i GDB dovesti sustav u neku funkciju (primjerice `k_startup`) te ispisati sadržaje varijabli.

6. Prekidi

Upravljanje periferijom[1] – ulazno/izlaznim napravama svodi se na slanje upravljačkih naredbi i podataka izravno preko mapiranih spremničkih lokacija ili posebnim instrukcijama. Upravljanje obavlja procesor prikladnim instrukcijama upravljačkih programa (engl. *device driver*). Međutim, kada naprava obavi svoj posao ili ako se pojavi događaj koji zahtjeva hitnu obradu (od strane procesora) tada naprava o tome obavještava procesor korištenjem mehanizma *prekida*.

Svaka naprava je spojena na prekidni ulaz procesora (u raznim arhitekturama može biti i više ulaza ili se prekidni signali dovode do posebnog sklopa koji onda prosljeđuje zahtjev – prekid prema procesoru). Uobičajeno ponašanje procesora na zahtjeve za prekid koji su generirani od naprava – *sklopovski prekidi* (uzrok je izvan procesora) sastoji se u tome da se najprije dovrši tekuća instrukcija, a na njenom se završetku provjerava stanje prekidnog ulaza procesora. Ako je prekidni signal postavljen i trenutno je dozvoljeno prihvaćanje prekida (označeno posebnim bitom u registru stanja procesora), tada se prekid *prihvaća*. Način prihvaćanja prekida može biti donekle različit od arhitekture do arhitekture, ali neki osnovni koncepti su uglavnom slični.

U postupku prihvata prekida (nakon zahtjeva za prekid, nakon dovršetka tekuće instrukcije, uz dozvoljeno prekidanje) procesor reagira na sljedeći način:

1. zabranjuje daljnje prekidanje (primjerice briše odgovarajuću zastavicu u statusnom registru);
2. prebacuje se u *prekidni način rada* (u tom načinu se možda i automatski aktiviraju neki zamjenski registri procesora, omogućujući mu *privilegiraniji način rada*, iako za jednostavnije procesore ne mora postojati više načina rada, već se sve može obavljati u istome načinu);
3. pohranjuje se *programsko brojilo* na stog (barem programsko brojilo, najčešće još i registar stanja; to sada može biti i drugi stog, stog prekidnog načina rada) te
4. u programsko se brojilo stavlja adresa potprograma za obradu prihvaćenog prekida – adresa *prekidnog potprograma*.

Određivanje adrese prekidnog potprograma se u različitim arhitekturama radi na ponešto različite načine. Najčešće se uz prekid veže i *prekidni broj* (razni uzročnici prekida izazivaju zahtjeve za prekid s različitim brojevima, iako je negdje moguće da se prekidni brojevi i dijele između skupine naprava). Adresa potprograma određuje se preko prekidnog broja. Primjerice, u nekim arhitekturama se pri inicijalizaciji sustava na adresu 0 stavlja adresa prekidnog programa za prekid 0, na adresu 1 adresa za obradu prekida 1 i slično ili se na te adrese postavljaju instrukcije skoka na prekidne potprograme.

Prekid se javlja u "nepredviđenom" trenutku. Procesor je u tom trenutku radio nešto – izvodio neki program – *dretvu*[2] tog programa. Prihvatom prekida privremeno je prekinuto izvođenje te dretve. Sustav prihvata i obrade prekida zato treba omogućiti da se po završetku obrade prekida sustav vraća u prekinutu dretvu i normalno nastavlja s njenim radom. Mehanizam koji to omogućava je pohrana *konteksta dretve* (sadržaja svi relevantnih registara procesora koji sadrže podatke koje dretva koristi) prije početka obrade prekida te obnova *konteksta* prije povratka iz prekida. Spremanje konteksta je zapravo prva aktivnost koja se izvodi na početku potprograma za obradu prekida, a obnova konteksta je zadnja aktivnost prekidnog potprograma. Uz obnovu konteksta prekinute dretve procesor se mora vratiti u način rada prekinute dretve te ponovno dozvoliti prekidanje. Ovih zadnjih par aktivnosti se najčešće izvodi jednom instrukcijom za povratak iz prekida.

[1] Periferija u ovom kontekstu predstavlja skoro sve komponente računalnog sustava, izuzev procesora i spremnika!
[2] Pojam *dretva* je detaljnije opisan u 11. poglavlju. Do tada se on može poistovjetiti s pojmom *program u izvođenju*.

Osim "vanjskih" sklopovskih prekida, i sam procesor u izvođenju instrukcija može izazvati prekide. Primjeri za to su pokušaj dijeljenja s nulom, pokušaj izvođenja (dekodiranja) nepostojećeg instrukcijskog kôda, pokušaj pokretanja nedozvoljene instrukcije (zbog nedostatnih privilegija trenutne dretve), pokušaj dohvata podatka ili instrukcije s nepostojeće (ili nedozvoljene) spremničke lokacije i slično. Takve prekide možemo nazvati *iznimkama* (engl. *exception*).

Procesori imaju i posebnu instrukciju (ili više njih) za izazivanje *programskih prekida*. Programskim se prekidom "poziva" zaštićen i privilegiran potprogram koji obavlja posebne aktivnosti u korist pozivajuće dretve (dretva je na taj način pozvala *jezgrenu funkciju*).

Većina procesora jednako reagira na prekid, neovisno od kuda on potiče (izvan ili u procesoru).

Upravljanje prekidnim podsustavom sastoji se od inicijalizacije prekidnog podsustava, definiranje ponašanja za pojedine prekida, ali i omogućavanje dinamičkog definiranja ponašanja za neke prekide, primjerice za potrebe nekih upravljačkih programa.

6.1. Prekidni sustav arhitekture x86

Izvore prekida u arhitekturi x86 [Intel, 2009] dijelimo u tri skupine:

1. prekide izazvane izvan procesora – sklopovski prekidi
2. prekide izazvane u procesoru uslijed problema s izvođenjem instrukcije – iznimke
3. prekide izazvane instrukcijom za izazivanje prekida – programski prekidi.

Prekidi s uzrokom izvan procesora se do procesora ne prenose izravno od naprave koja je prekid izazvala, već preko zasebnog međusklopa – upravljača prekida (engl. *programmable interrupt controller – PIC*). Taj se međusklop može programirati tako da neke prekide propušta do procesora, a neke ne (ili barem privremeno ne). Detaljniji prikaz sklopa za upravljanje prekidima *Intel 8259* dan je u odjeljku 6.3.

Prekidi izazvani unutar procesora najčešće označavaju nenormalno stanje dretve, koju tada treba prekinuti (izuzeci su prekidi zbog straničenja).

Bez obzira na uzrok prekida, ponašanje procesora pri prihvatu prekida je isto. Neka za početak procesor ostaje u istom stanju i u obradi prekida, tj. već se nalazi u privilegiranom načinu rada i ne prelazi iz korisničkog načina rada u prekidni način rada (već je u tom načinu rada). Razmatranje ovih drugih slučajeva odgođeno je za kasnije (poglavlje 12.).

U razmatranom slučaju pri prihvatu prekida događa se niz aktivnosti:

- na stog se postavljaju redom (od dna prema vrhu stoga):
  - i) registar stanja – pohranjuju se zastavice (`eflags`);
  - ii) registar oznake segmenta instrukcija (`cs`, detaljnije o ovom registra u odjeljku 12.5.1.);
  - iii) programsko brojilo (registar `eip`);
  - iv) neki prekidi (iznimke) još na stog stavljaju i kôd greške;
- na temelju uzročnika prekida (prekidnog broja) određuje se adresa prekidnog potprograma:
  - i) pri tome se koristi posebna tablica opisnika prekida naziva *IDT* (kratica od *interrupt descriptor table*) koja sadrži adrese prekidnih potprograma (ali i još neke podatke, za sada nevažne);
  - ii) na temelju prekidnog broja dolazi se do odgovarajućeg retka tablice *IDT* iz kojeg se učitava adresa prekidnog potprograma u programsko brojilo;
- iduća instrukcija pripada prekidnom potprogramu.

Za povratak iz prekida koristi se posebna instrukcija `iret` koja na stogu očekuje redom od vrha prema dnu:

1. povratnu adresu (koja je spremljena pri prihvatu prekida),
2. registar oznake segmenta instrukcije te
3. registar stanja.

Ako se u obradi prekida nešto postavljalo na stog to se sve mora maknuti (uključujući eventualno i kôd greške).

Ponašanje procesora prema (sklopovskim) prekidima definirano je stanjem bita `IF` u registru stanja (`eflags`). Slika 6.1. (preuzeta iz [Intel, 2009]) prikazuje sve zastavice registra stanja među kojima je i zastavica `IF`.

Slika 6.1. Registar stanja (`eflags`)

Zastavicom `IF` utječe se samo na sklopovske prekide – prekide koji se generiraju izvan procesora (*maskirajuće prekide*). Prekidi generirani u procesoru uslijed izvođenja instrukcija ne mogu se zabraniti (to su tzv. *nemaskirajući prekidi*).

Instrukcije kojima se utječe na prihvat sklopovskih prekida, same izazivaju prekide i slično, uključuju:

- `cli` – briše zastavicu `IF` – onemogućava sklopovske prekide (engl. *clear interrupt flag*)
- `sti` – postavlja zastavicu `IF` – omogućava sklopovske prekide (engl. *set interrupt flag*)
- `int n` – programsko izazivanje prekida s brojem `n` (engl. *call to interrupt procedure*)

- `iret` – povratak iz prekidnog potprograma (engl. *interrupt return*).

6.2. Upravljanje prekidima

Od podsustava za upravljanje prekidima očekuje se da omogući povezivanje prekida s odgovarajućim upravljačkim programima te da obavlja sve međukorake koji su potrebni pri prihvatu prekida, a prije poziva prekidnog potprograma, kao i po dovršetku obrade prekida, a prije povratka u prekinuti program (dretvu). Također, od podsustava se očekuje sučelje za omogućavanje i onemogućavanje prekidanja procesora od strane raznih naprava pa i svih u nekim trenucima.

Upravljanje prekidom u Benu[3] ostvareno je u sloju *arch*.

Osnovna ideja ostvarenja jest da se sve obrade preusmjere na zajedničku funkciju (u ovom primjeru na `arch_interrupt_handler`) u kojoj se mogu raditi dodatne provjere te od tu pozvati registriranu funkciju koja će obraditi prekid. Stoga je tablica IDT početno inicijalizirana (u `arch/i386/descriptors.c`) kazaljkama na funkcije koje obavljaju početni prihvat prekida (oblika `interrupt_n`, ostvarene u `arch/i386/interrupts.S`), ali prosljeđuju obradu u funkciju `arch_interrupt_handler`. S obzirom na to da se prekidni broj ne može doznati na drugi način, za svaki prekid treba napraviti zasebnu funkciju u `interrupts.S` koja će pri pozivu `arch_interrupt_handler` kao argument poslati prekidni broj.

Osnovne funkcije, tj. sučelja (definirana u `include/arch/interrupt.h`) su:

- `arch_init_interrupts` – inicijalizira strukturu podataka koja povezuje prekide s upravljačkim programima te
- `arch_register_interrupt_handler` – povezuje zadani prekid s funkcijom za obradu, tj. s upravljačkim programom.

6.2.1. Datoteka `interrupt.S`

Početne funkcije za prihvat prekida za sve prekide definirane su u datoteci `interrupts.S` u sloju *arch*. S obzirom na to da je obrada slična za svaki prekidni potprogram, nakon par instrukcija prelazi se na zajednički dio.

U prvom koraku (`01_Exceptions`) je svaka funkcija, za svaki prekidni broj, zasebno navedena. Primjer funkcije za prekid 0 je naveden u nastavku.

Isječak kôda 6.1. Chapter_03_Interrupts/01_Exceptions/arch/i386/interrupt.S

```
24 interrupt_0:
25         pushl   $0              /* dummy error code when real is not provided */
26         pushal                  /* save 'context' (general registers) */
27
28         pushl   $0              /* push interrupt number on stack */
29         call    arch_interrupt_handler
30         addl    $4, %esp        /* remove interrupt number from stack */
31
32         popal                   /* restore 'context' */
33         addl    $4, %esp     /* remove error code (real or dummy) from stack */
34
35         iret    /* return from interrupt to thread (restore eip, cs, eflags) */
```

U ostalim koracima (i dalje), umjesto da je svaka funkcija zasebno navedena, korišteni su makroi koji će prije prevođenja "proširiti" datoteku tako da onda postoje sve navedene funkcije.

[3] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_03_Interrupts`.

Isječak kôda 6.2. Chapter_03_Interrupts/02_PIC/arch/i386/interrupt.S

```
24 .irp int_num,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,\
25         25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48
26 .type interrupt_\int_num, @function
27
28 interrupt_\int_num:
29
30 .if \int_num < 8 || \int_num == 9 || \int_num > 14
31         pushl   $0              /* dummy error code when real is not provided */
32 .endif
33
34         pushal                  /* save 'context' (general registers) */
35
36         pushl   $\int_num       /* push interrupt number on stack */
37
38         call    arch_interrupt_handler
39
40         addl    $4, %esp        /* remove interrupt number from stack */
41
42         popal                   /* restore 'context' */
43
44
45         addl    $4, %esp     /* remove error code (real or dummy) from stack */
46
47         iret    /* return from interrupt to thread (restore eip, cs, eflags) */
48 .endr
```

Makro `.irp` definira dio kôda koji će se ponavljati (pri generiranju kôda ovaj će se dio uvišestručiti) i u svakoj kopiji u varijablu `int_num` (prekidni broj) stavit će jednu od vrijednosti navedene iza varijable. Definirani su prekidni potprogrami za prvih 49 prekida (0-48).

Oznaka `interrupt_\int_num` će u procesu prevođenju se najprije zamijeniti s `interrupt_0` , `interrupt_1`, ... za sve zadane vrijednosti za `\int_num`. Te oznake predstavljaju adrese prekidnih potprograma te su kasnije posložene u jedno polje koje se koristi pri inicijalizaciji prekidnog podsustava (tablice *IDT*).

Kako neki prekidi na vrh stoga dodatno postavljaju kôd greške i za preostale je prekide na stog postavljen dodatni podatak (nula) kao zamjenski kôd greške tako da je stog identičan za sve prekide (linije 30-32).

Instrukcija `pushal` spremit će sve registre opće namjene na stog (registre: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, i `edi`).

Nakon minimalnog konteksta i registara opće namjene na stog se pohranjuje i prekidni broj (identifikator prekida). Prema ovom broju identificira se uzrok prekida i kasnije poziva odgovarajući potprogram za njegovu obradu. Budući da je on pohranjen na stog on je ujedno i prvi (i za sada jedini) parametar funkcije `arch_interrupt_handler`. Nakon obrade prekida (završetka prethodne funkcije), treba redom:

1. s vrha stoga maknuti prekidni broj
2. obnoviti kontekst prekinute dretve (registre opće namjene)
3. sa stoga još skinuti kôd greške
4. vratiti se u prekinutu dretvu.

Sve navedeno obavlja niz instrukcija u linijama 40-47.

Makro `.irp` završava oznakom `.endr`. Korištenjem navedenog makroa, za svaki od navedenih brojeva prekida (od 0 do 48) prije prevođenja generirat će se isti kôd, uz zamjenu `\int_num` s odgovarajućim brojem.

Isječak kôda 6.3. Chapter_03_Interrupts/02_PIC/arch/i386/interrupt.S

```
53 /* Interrupt handlers function addresses, required for filling IDT */
54 .type   arch_interrupt_handlers, @object
55 .size   arch_interrupt_handlers, 49*4
56
57 arch_interrupt_handlers:
58 .irp int_num,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,\
59         25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48
60
61         .long interrupt_\int_num
62 .endr
63         .long 0
```

Na kraju datoteke u jedno polje stavljene su adrese svih prekidnih potprograma da bi se mogao postaviti prekidni podsustav (funkcija `IDT_init` u datoteci `descriptor.c`). Kao i za samo definiranje funkcija i za ovo je korišten makro `irp`.

Posebnim označavanjem prekidnih funkcija prekidni potprogrami se mogu napisati i u C-u. Npr. kada se koristi GCC prekidne funkcije se mogu pripremiti kao u sljedećem primjeru.

Primjer prekidne funkcije u C-u

```
struct okvir_prekida { /* za i386 */
    uword_t ip;
    uword_t cs;
    uword_t flags;
};
__attribute__ ((interrupt))
void obrada_prekida(struct okvir_prekida *okvir_prekida)
{
    /* tijelo funkcije za obradu prekida */
}
```

Za gornji primjer potrebno je dodati i zastavicu `-mgeneral-regs-only`. Ovakvo ostvarenje funkcija za obradu prekida pokazano je samo u pomoćnom koraku `01x_Exceptions`. Više o takvim funkcijama na [gcc-interrupt].

6.2.2. Datoteka `interrupts.c`

U datoteci `arch/i386/interrupts.c` nalaze se osnovne funkcije za upravljanje prekidnim podsustavom te podatkovna struktura koja povezuje registrirane prekide s njihovim funkcijama.

Isječak kôda 6.4. Chapter_03_Interrupts/01_Exceptions/arch/i386/interrupt.c

```
10 /*! interrupt handlers */
11 static void (*ihandler[INTERRUPTS])(unsigned int);
```

Funkcija `arch_register_interrupt_handler(inum, handler)` povezuje određeni prekid (prekidni broj) s funkcijom za obradu tog prekida.

Isječak kôda 6.5. Chapter_03_Interrupts/01_Exceptions/arch/i386/interrupt.c

```
49 /*!
50  * "Forward" interrupt handling to registered handler
51  * (called from interrupts.S)
52  */
53 void arch_interrupt_handler(int irq_num)
54 {
55         if (irq_num < INTERRUPTS && ihandler[irq_num])
56         {
57                 /* Call registered handler */
58                 ihandler[irq_num](irq_num);
59         }
60         else {
```

```
61                  LOG(ERROR, "Unregistered interrupt: %d !\n", irq_num);
62                  halt();
63          }
64  }
```

Po pojavi prekida prvo se poziva odgovarajuća funkcija iz `interrupts.S` (`interrupt_n`), a iz nje funkcija `arch_interrupt_handler`. Ako se radi o prekidu za koji je registrirana funkcija, daljnja obrada se prosljeđuje u tu funkciju (`ihandler[irq_num](irq_num)`).

Korištenje tablice *IDT* implicitno podrazumijeva i korištenje tablice *GDT*. Opis načela koja se koriste pri raznim operacijama nad toj tablici ostavljen je za kasnije (odjeljak 12.5.1.), kada se raspravlja o mogućnostima upravljanja spremnikom, kada program, tj. dretve rade u korisničkom načinu rada. Operacije za inicijalizaciju obiju tablica ostvarene su u datoteci `arch/i386/descriptors.c`.

6.2.3. Datoteka `kernel/startup.c`

Početna jezgrena funkcija u ovoj je fazi proširena pozivima inicijalizacije prekidnog podsustava te, radi ispitivanja rada podsustava, pozivom programa (funkcije) `segm_fault` koja izaziva programske prekide.

Isječak kôda 6.6. Chapter_03_Interrupts/01_Exceptions/kernel/startup.c

```
26  void k_startup()
27  {
```

Isječak kôda 6.7. Chapter_03_Interrupts/01_Exceptions/kernel/startup.c

```
37          /* interrupts */
38          arch_init_interrupts();
```

Isječak kôda 6.8. Chapter_03_Interrupts/01_Exceptions/kernel/startup.c

```
53          hello_world();
54          segm_fault();
```

Osim inicijalizacije prekidnog podsustava, u gornjem je primjeru još pokazano i kako se registrira pojedina funkcija za pojedini prekid.

Isječak kôda 6.9. Chapter_03_Interrupts/01_Exceptions/programs/segm_fault/segm_fault.c

```
13  static void test1(uint irqn)
14  {
15          printf("Interrupt handler routine: irqn=%d\n", irqn);
16  }
17
18  int segm_fault()
19  {
20          printf("\nInterrupt test >>>\n");
21
22          arch_register_interrupt_handler(SOFTWARE_INTERRUPT, test1);
23          arch_register_interrupt_handler(SOFTWARE_INTERRUPT, test1);
24
25          raise_interrupt(SOFTWARE_INTERRUPT);
26
27          printf("Interrupt test <<<\n\n");
28          return 0;
29  }
```

S obzirom na to da u ovom inkrementu još nije ostvaren niti jedan upravljački program, prekidi se izazivaju programski (`raise_interrupt`). Pri pojavi registriranog prekida poziva se registrirana funkcija i kao parametar joj se šalje brojčana oznaka prekida.

Postupak prihvata i obrade prekida u prikazanom primjeru može se prikazati detaljnije. Prihvat prekida dijeli se na dva dijela:

1. akcije koje poduzima sam procesor:
   - "osnovni kontekst" na stog (`EFLAGS`, `cs`, `eip`, kôd greške)
   - prema broju prekida određuje se prekidna funkcija:
     - broj određuje zapis *IDT* tablice => adresa prekidne funkcije
     - za `raise_interrupt (SOFTWARE_INTERRUPT)` (uz `SOFTWARE_INTERRUPT = 48` ⇒ broj = 48 ⇒ funkcija za obradu prekida je `interrupt_48`
2. obrada prekida – prekidni podsustav OS-a:
   `interrupt_48` ⇒ `arch_interrupt_handler` ⇒ `test1`.

6.3. Upravljanje prekidima sklopom *Intel 8259*

Sklop za upravljanje prekidima *Intel 8259* omogućuje programiranje prihvaćanja ili neprihvaćanja (maskiranja) zahtjeva za prekid koji dolaze od naprava koje su na njega spojene. Shema sklopa *Intel 8259* prikazana je na slici 6.2.

Slika 6.2. Sklop *Intel 8259*

Ulazi IR0–IR7 prate zahtjeve za prekid vanjskih naprava spojenih na njih. Ako je sklop programiran da određeni zahtjev proslijedi dalje on će to napraviti preko svog INT izlaza dok će na podatkovnoj sabirnici D0–D7 dati broj linije s koje prosljeđuje zahtjev.

Uobičajeno ostvarenje sklopa za upravljanje prekidima (PIC) kod arhitekture x86 sastoji se od dva takva sklopa: glavnog (engl. *master*) i pomoćnog (engl. *slave*). Glavni i pomoćni sklop povezani su u lanac tako da je pomoćni sa svojim INT izlazom spojen na IR2 ulaz glavnog sklopa. Na taj način sklopovi mogu prihvaćati 15 vanjskih zahtjeva za prekid, prema tablici 6.1.

Svaki se priključak zasebno može omogućiti ili onemogućiti, tj. propustiti zahtjev za prekid prema procesoru ili zaustaviti zahtjev. Izvorno, sklop izaziva prekide s brojevima od 0 do 15, ali se ti brojevi preklapaju s prekidima koje izaziva sam procesor. Zato se sklop može programirati da mu prekidi počinju od nekog drugog broja (broj prekida se prosljeđuje procesoru preko zasebne sabirnice (D0-D7). Zadnji stupac gornje tablice prikazuje uobičajeni odmak prekida, tj. brojeve prekida za pojedine naprave (koji su korišteni i ovdje). Više detalja o sklopu, načinu

Tablica 6.1. Uobičajeni priključci na *PIC*

Priključak	Spojeni uređaj	Broj prekida[a]
M-IR0[b]	sat (i8253)	32
M-IR1	tipkovnica	33
M-IR2	pomoćni sklop	34
M-IR3	serijski port (COM2)	35
M-IR4	serijski port (COM1)	36
M-IR5	disk ili paralelni port LPT2	37
M-IR6	disketna jedinica	38
M-IR7	paralelni port LPT1	39
S-IR0	sat	40
S-IR1	–	41
S-IR2	–	42
S-IR3	–	43
S-IR4	PS/2 miš	44
S-IR5	koprocesor	45
S-IR6	disk	46
S-IR7	disk	47

[a] Brojevi nakon programiranja sklopa tako da je prvi broj 32.

[b] M označava ulaze glavnog sklopa, S pomoćnog.

programiranja i slično može se naći u literaturi i na Internetu [Intel 8259].

Inicijalizacija sklopa uključuje već opisani pomak u brojevima generiranih prekida te početno maskiranje (onemogućavanje) svih prekida. Osim inicijalizacije, za upravljanje sklopom potrebne su funkcije koje će dozvoliti ili zabraniti prekid pojedinog priključka.

Sklop se počinje koristiti u drugoj fazi inkrementa (03/02) kôdom iz `arch/i386/drivers/i8259.c`. Kôd ostvaruje sučelje `arch_ic_t` definirano posebno za sklopove koji imaju mogućnost upravljanja prekidima naprava (`arch/i386/interrupts.h`).

Osim inicijalizacije, sučelje definira mogućnost zabrane ili omogućavanja pojedinog prekida (koji može definirati primjerice redni broj naprave spojene preko tog sklopa). Ponekad je nakon prekida potrebno obaviti dodatne poslove te je i za to predviđena funkcija (`at_exit`). Na primjer, sklopu *Intel 8259* je po obradi sklopovskog prekida potrebno poslati posebnu oznaku `EOI` (skraćenica od *end of interrupt*), da može propuštati iduće prekide.

Isječak kôda 6.10. Chapter_03_Interrupts/02_PIC/arch/i386/interrupt.h

```
16 /*! (Hardware) Interrupt controller interface */
17 typedef struct _interrupt_controller_
18 {
19         void  (*init)();
20         void  (*disable_irq)(unsigned int irq);
21         void  (*enable_irq)(unsigned int irq);
22         void  (*at_exit)(unsigned int irq);
23
24         char  *(*int_descr)(unsigned int irq);
25 }
26 arch_ic_t;
```

Sučeljem `arch_irq_enable (irq_num)` omogućava se prekid određene naprave (koja izaziva prekid broja `irq_num`), dok će se sučeljem `arch_irq_disable (irq_num)` to onemogućiti.

Ako se u nekom trenu žele zabraniti svi prekidi onda treba koristiti posebnu instrukciju procesora (`cli`), koja se preko sučelja može pozvati s `disable_interrupts` (`arch/i386/processor.h`).

6.4. Ostale mogućnosti upravljanja prekidima

6.4.1. Upravljanje prekidima na nižoj razini

U prikazanom ostvarenju radi jednostavnosti svi se prekidi iz početnih funkcija preusmjeruju na `arch_interrupt_handler`. U jednostavnijim sustavima bi se i to moglo izbjeći i sve ostvariti korištenjem asemblera. Ipak, s obzirom na to da je to značajno teže (a i kôd više nije "prenosiv") treba dobro razmisliti o tom izboru, pogotovo stoga što današnji prevoditelji mogu poprilično dobro optimirati i prema kriteriju brzine rada, ali i prema kriteriju veličine programa. Alat *GCC* ima nekoliko zastavica koje utječu kriterije optimiranja:

- `-O`, `-O1`, `-O2`, `-O3` – optimiranje obzirom na brzinu (primjerice `-O3` će pored ostalog i sam pokušati odlučiti koje funkcije može koristiti kao `inline`, bez obzira na to što nemaju te oznake)
- `-Os` – optimiranje s obzirom na veličinu programa (što je moguće manji)
- druge zastavice (pogledati odjeljak *Optimize Options* u [GCC]).

6.4.2. Problem dugotrajnih prekidnih potprograma

Jedan od problema s obradom prekida na prikazani način u Benu jest da se obrada u cijelosti obavlja u prekidnom načinu rada, u kojem su prekidi zabranjeni do završetka obrade prethodnog prekida. Ako bi neka obrada potrajala, svi zahtjevi za prekid koji dođu u međuvremenu moraju čekati. Takva duža zadržavanja obrade prekida ponekad nisu prihvatljiva. Načini rješavanja ili barem ublažavanja ovog problema su razni.

Jedno rješenje može uključivati dodjelu prioriteta prekidima te obavljanje obrade uz dozvoljeno prekidanje. Upravljanje prioritetima može se riješiti i sklopovski, ako postoji takva podrška, ali može i programski. Programsko rješenje bi tražilo proširenje kôda kućanskih poslova, gdje bi trebalo dodati kôd koji utvrđuje uzročnika prekida i njegov prioritet. U ovisnosti o njemu treba odmah započeti obradu novog prekida, ako je njegov prioritet veći od prioriteta trenutno prekinute dretve ili prioriteta obrade prekida ili odgoditi obradu dok se prioritetniji poslovi ne obave. Navedeno rješenje je detaljnije prikazano u [Budin, 2010].

Drugo rješenje uključuju podjelu posla obrade prekida na dva dijela. U prvom dijelu koji se

poziva u trenutku prihvata prekida, napravi se samo osnovna obrada (primjerice pohranjuju potrebni podaci). Drugi dio obrade se obavlja naknadno, u skladu s prioritetom. Jedan od načina odrade drugog dijela prekida može biti korištenje (prioritetnog) reda u koji se svrstavaju zahtjevi za obradu. Jedna ili više dretvi može kasnije obaviti posao do kraja. Ili se za svaku obradu prekida može stvoriti nova dretva koja će obraditi taj drugi dio posla (ili se umjesto skupe operacije stvaranja nove dretve može uzeti neka iz skupine već stvorenih dretvi i predviđenih za ovu namjenu).

U Benu se ne koriste posebni mehanizmi rješavanja problema dugotrajnih obrada prekida. Međutim, kasnije će se prikazati mehanizmi za upravljanje dretvama i njihovu sinkronizaciju pomoću kojih se može posao obrade podijeliti na dva dijela: u prvome se mogu pohraniti potrebni podaci za obradu te potom aktivirati dretva koja čeka na takve zahtjeve, koja će tada obaviti drugi dio obrade prekida.

6.4.3. Dijeljenje prekidne linije

Ako istu prekidnu liniju (isti broj) dijeli više uređaja, prikazani podsustav za upravljanje prekidima bi trebalo proširiti. U kasnijim je koracima taj problem riješen na način se za svaki mogući prekidni broj koristi lista registriranih funkcija. Kada se prekid dogodi pozivaju se redom sve registrirane funkcije. Te funkcije treba napraviti tako da one na početku svog izvođenja dodatno provjere je li zaista potrebno njihovo izvođenje, tj. prvo trebaju provjeriti statusne registre naprava koje poslužuju. Ako te naprave nisu izazvale prekid, funkcije završavaju te se pokreće iduća u nizu.

Da bi se navedeni način preko liste registracija za pojedini prekid mogao ostvariti potrebni su mehanizmi *dinamičkog upravljanja spremnikom* koji će omogućiti dinamičko stvaranje objekta registracije za pojedini prekid i njegovo stavljanje u listu za odabrani prekid. Dinamičko upravljanje spremnikom potrebno je i za druge objekte koji se koriste u jezgri operacijskog sustava te je njemu posvećena posebna pažnja (u sljedećem poglavlju).

6.4.4. Sporadični prekidi

Ponekad se signal prekida pojavljuje iako ga niti jedna naprava nije namjerno izazvala (engl. *spurious interrupt*). Uzroci mogu biti razni, a najčešće su nekakve kratkotrajne električne smetnje. S obzirom na to da će procesor detektirati takav signal i uobičajenim putem pozvati proceduru za obradu, poželjno je da svaka takva procedura, tj. upravljački program, na svom početku prvo provjeri je li njegovo pokretanje zaista opravdano. Najčešće se to može ustanoviti ispitivanjem statusnih registara dotične naprave. Ako ona nije izazvala prekid, prava obrada se ne poziva i procesor se vraća prekinutom programu (dretvi).

6.4.5. Vrlo brze naprave

Neke naprave kao što je su to grafička i mrežna kartica mogu raditi vrlo brzo te stoga i često tražiti suradnju procesora. Iako je prekid uobičajeni način komunikacije naprava s procesorom, svaka obrada prekida osim korisnog rada uključuje i kućanske poslove – promjenu načina rada procesora, spremanje kontesta te obnavljanje konteksta i promjenu načina rada procesora po dovršetku obrade. Navedeni kućanski poslovi mogu i na modernijim računalima potrajati i do jedne mikrosekunde. Kada su prekidi rijetki (s intervalima bar za red veličine većima od trajanja kućanskih poslova) tada je njihov utjecaj na sustav gotovo neprimjetan. Međutim, kada bi prekidi bili češći, tada bi učinkovitost sustava značajno pala jer bi primjetan dio procesorskog vremena bio potrošen na kućanske poslove koji nisu nikakav produktivan rad. Stoga se u mnogim modernim sustavima prati učestalost pojave prekida naprava. Kada učestalost pređe definiranu granicu, napravama se zabrane prekidi i one se poslužuju izravno, s definiranim intervalima. Pretpostavka je da će takva naprava moći sve svoje operacije zadržati tako da

se obave odjednom, kada dobiju procesorsko vrijeme. Primjerice, kada mrežna kartica treba poslati puno podataka velikom brzinom, procesor može sam periodički puniti međuspremnik te naprave, bez da ona traži prekid nakon svakog poslanog paketa. Na taj način jedna naprava neće zauzeti svo procesorsko vrijeme – operacijski sustav će odrediti koji dio vremena može odvojiti za posluživanje naprave. Tako se ostvaruje mogućnost boljeg upravljanja sustavom i kvalitetom usluge koju operacijski sustav pruža pojedinim programima i napravama (engl. *quality of service – QoS*).

Pitanja za vježbu 6

1. Čemu služe prekidi?
2. Opisati postupak prihvata prekida te povratka iz prekida.
3. Zašto se pri prihvatu prekida na stog pohranjuje i registar stanja i programsko brojilo?
4. Što su to sklopovski prekidi, iznimke i programski prekidi? Tko ih izaziva?
5. Što je to prekidni broj?
6. Što su to maskirajući, a što nemaskirajući prekidi?
7. Ako je zastavica `IF` (engl. *interrupt flag*) obrisana, hoće li instrukcija `INT 33` izazvati prekid (koji će se prihvatiti i obraditi)? Zašto?
8. Tri osnovne funkcije prekidnog sustava zadanog projekta su: `arch_init_interrupts`, `arch_interrupt_handler` i `arch_register_interrupt_handler`. Što one rade?
9. Koja je zadaća podsustava za upravljanje prekidima?
10. Koja je zadaća sklopa za upravljanje prekidima?
11. Obrada prekida može potrajati. Zašto to ponekad može predstavljati veliki problem? Kako se problem može riješiti ili ublažiti?
12. Koji problemi mogu nastati pri obradi prekida ako se prekidi jako često pojavljuju?

7. Algoritmi upravljanja spremnikom

7.1. Statičko upravljanje spremnikom

Operacijski sustav za svoj rad i upravljanje treba određenu strukturu podataka. Neki od tih podataka su zapravo globalne varijable te je u spremniku mjesto za njih odmah po pokretanju zauzeto. Primjere ovakvih struktura podataka možemo vidjeti gotovo u svakom podsustavu jezgre, na primjer, varijable `k_stack`, `u_stdout`, `k_stdout`, `icdev` i `ihandler`. Takve varijable zauzimaju spremnički prostor od njegova pokretanja do njegova gašenja. One su *statički* zauzele spremnički prostor te bi takav implicitni način upravljanja spremnikom mogli nazvati *statičkim upravljanjem spremnikom*.

Statičko upravljanje je dovoljno samo za jedan manji dio strukture podataka. Naime, tijekom rada pojavljuju se zahtjevi za određenim operacijama koje će privremeno zahtijevati određeni blok spremnika za privremenu pohranu podataka i međurezultata. Ponekad će te operacije stvoriti objekte koji bi trebali duže ostati u sustavu. U jednostavnijim sustavima moguće je možda predvidjeti takve zahtjeve i statički zauzeti spremnički prostor za to.

Primjerice, ako je moguće procijeniti dovoljnu veličinu za stog on se može statički zauzeti (kao što je u početnim inkrementima Benua). Slično se može napraviti i za druge elemente sustava, kao što su opisnici za registraciju prekida, opisnici za alarme, korištenje naprava, opisnici za dretve i slično. Ako je unaprijed poznat broj takvih elemenata, onda se oni mogu zauzeti u samom kôdu kao varijable i polja. Pri svakom novom zahtjevu za takav element može se pretraživati zadano polje u potrazi za slobodnim elementom i njega iskoristiti u novom zahtjevu.

Prikazano statičko rješenje može biti vrlo jednostavno. Međutim, za neke primjene slijedno pretraživanje nije odgovarajuće jer je $O(N)$ složenosti što ne zadovoljava zahtjeve sustava za rad u stvarnom vremenu. Nadalje, ako bi se i za druge elemente sustava koristila slična načela, već bi u početku velik dio spremnika bio statički zauzet za te strukture podataka. Opravdanost ovakvog načina uvelike bi ovisila i o procjeni potrebne veličine pojedinih struktura podataka. Ako se zauzme premalo, sustav neće moći obavljati zadanu mu funkciju. Ako se zauzme previše, trebat će veći spremnički prostor, što diže cijenu sustava. Negdje to može biti prihvatljivo, ali u većini slučajeva nije.

7.2. Dinamičko upravljanje spremnikom

Pokazalo se da sustav u raznim trenucima različito koristi svoja sredstva. U jednom trenutku je aktivniji u jednom podsustavu (primjerice mrežnom), a u drugom trenutku drugi je podsustav aktivniji i tada treba više spremničkog prostora. U takvom dinamičkom okruženju može se i s manjim spremnikom poslužiti zahtjeve. Umjesto da se dijelovi spremnika statički zauzmu za sve moguće buduće potrebe, on se može dodjeljivati dinamički, prema zahtjevima koji se javljaju tijekom rada sustava. Pri pokretanju sustava za takve dinamičke zahtjeve zauzima se dio spremnika koji se naziva *gomila* (engl. *heap*). Tijekom rada sustava gomila se prikladnim algoritmom dijeli na blokove (zauzete i slobodne), prema zahtjevima za spremnikom. U nekom trenutku slika gomile može izgledati kao na slici 7.1.

Osnovna pretpostavka ovakvog načina upravljanja spremnikom jest da će se nakon završetka korištenja dodijeljenih blokova isti vratiti sustavu i na taj način moći ponovno iskoristiti za neke druge buduće zahtjeve.

Način dodjele dijelova gomile ovisi o korištenom algoritmu *dinamičkog upravljanja spremnikom*. Algoritam definira način podjele spremnika, način traženja bloka koji odgovara zahtjevima te način organizacije *slobodnih blokova* spremnih za dodjelu. Pri odabiru odgovarajućeg algoritma

Slika 7.1. Primjer gomile tijekom rada, podijeljene na slobodne i zauzete blokove

treba uzeti u obzir neka njihova svojstva:

- složenost dodjele i oslobađanja (koliko će to trajati?)
- utjecaj fragmentacije (moguće iskorištenje spremničkog prostora)
- svojstva zahtjeva (zahtjevi za velikim ili malim blokovima ili su zahtjevi slični, utjecaj zaglavlja blokova).

Najjednostavniji algoritmi u ostvarenju koriste *liste*. Slobodni blokovi (spremni za dodjelu) spremaju se u jednu listu. Prije stavljanja bloka u listu slobodnih blokova, blok se najprije pokušava spojiti sa susjednim blokovima, ako je ikoji od njih slobodan. Lista slobodnih blokova ne mora biti složena po nekom kriteriju, ali i može.

Na primjer, lista može biti uređena prema veličini blokova omogućujući dodjeljivanje najmanjeg slobodnog bloka koji je dovoljno velik za traženi zahtjev, ostvarujući metodu *najbolji odgovarajući* (engl. *best-fit*), ali će to povećati složenost ubacivanja slobodnog bloka u listu.

Drugo načelo uključuje korištenje neuređene liste, gdje se ona slijedno pretražuje te se uzima prvi pronađeni blok koji je odgovarajuće veličine (jednake ili veće od zahtjeva) – *prvi odgovarajući* (engl. *first-fit*). Ako se oslobođeni (slobodni) blokovi stavljaju na kraj liste, onda se načelo može nazvati *po redu prispijeća* (engl. *first-in-first-out – FIFO*), u suprotnome, ako se oslobođeni blokovi stavljaju na početak liste koristi se pojam *obrnuti red prispijeća* (engl. *last-in-first-out – LIFO*).

Svaki od navedenih (i nenavedenih) postupaka ima svoje prednosti i nedostatke. Primjerice, načelo obrnutog reda prispijeća omogućava veću iskoristivost priručnog spremnika procesora (i time ubrzava izvođenje), ali i značajno povećava fragmentaciju blokova.

Osim spomenutih metoda, u stvarnim ostvarenjima se koriste i druge. Jedan od poznatijih algoritama je *dlmalloc* (*Doug Lee's malloc*) [dlmalloc] koji slobodne blokove svrstava prema veličini u različite liste, prema slici 7.2. Manje blokove smješta u liste koje sadrže blokove identičnih veličina, dok su veći blokovi u listama približno istih veličina (te su liste uređene prema veličini).

Jedan od često korištenih algoritama je i *Buddy* algoritam, pogotovo za ugrađene sustave. Algoritam dijeli blokove samo u veličine koje su potencije broja 2. Spajanje je moguće samo ako su dva bloka susjedna i istih veličina (buddy blocks). Spajanje i odvajanje u takvom je sustavu jednostavno ostvariti i logaritamske je složenosti ($O(logN)$). Primer rada algoritma prikazan je na slici 7.3.

Jedan od problema *Buddy* algoritma kad bi se koristio kao osnovni algoritam upravljanja spremnikom bila bi *unutarnja fragmentacija*. Naime, ako se za, primjerice gornji primjer zatraži blok od 129 KB, algoritam će dodijeliti 256 KB, odnosno 127 KB više nego je traženo i potrebno. Tih 127 KB neće biti korišteno i neće se moći iskoristiti od strane drugih zahtjeva.

Korištenje priručnog spremnika procesora je vrlo bitno želi li se postići velika učinkovitost. Zato "moderniji" postupci upravljanja spremnikom su upravo tome usmjereni. Jedan od uobičajenih postupaka jest da se raspoloživi spremnik podijeli u nekoliko segmenata, od kojih će svaki biti korišten za određene tipove zahtjeva, najčešće rangiranih prema veličini ili učestalosti zahtjeva ili oboje). Jedan od takvih je i *slab cache* algoritam (ili *slab allocator*) koji uzima u obzir i raspo-

Slika 7.2. Liste slobodnih blokova kod *dlmalloc* algoritma

	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K	64K
t=0	1024K															
t=1	A	64K	128K		256K				512K							
t=2	A	64K	B		256K				512K							
t=3	A	C	B		256K				512K							
t=4	A	C	B		D		128K		512K							
t=5	A	64K	B		D		128K		512K							
t=6	128K		B		D		128K		512K							
t=7	256K				D		128K		512K							
t=8	1024K															

Slika 7.3. Primjer rada *Buddy* algoritma

dijeljenost blokova po stranicama radnog spremnika (podjela koja se koristi kod straničenja).

U trećoj fazi trećeg inkrementa[1] ostvarene su dvije metode: *prvi odgovarajući* uz načelo obrnutog reda prispijeća (oslobođeni blok se stavlja na početak liste) te *dvorazinsko polje uređenih listi* – TLSF (engl. *two level segregated first*).

7.2.1. Metoda prvi odgovarajući

Prvi odgovarajući je vrlo jednostavna metoda upravljanja slobodnim blokovima kod koje se u potrazi za odgovarajućim slobodnim blokom slijedno pretražuje nesređena lista. Zbog toga je algoritam vrlo jednostavan, ali i može nepredviđeno dugo trajati ako je dovoljno velik slobodni blok tek pri kraju liste. Drugi nedostatak je u povećanoj fragmentaciji slobodnih blokova s obzirom na to da se uvijek dodijeljuje prvi odgovarajući blok koji je možda i značajno veći od traženog. Ako je blok veći onda se dijeli na dva dijela: jedan se dodijeljuje (prema zahtjevu) a drugi se vraća u listu slobodnih blokova. Ovakvom podjelom smanjuje se broj velikih blokova i time možda onemogućava neki budući zahtjev za većim blokom.

Dinamičko upravljanje spremnikom prema metodi prvi odgovarajući ostvareno je u `lib/mm/ff_simple.c` i sastoji se od tri osnovne funkcije (sučelja):

- `ffs_init(mem_segm, size)` – inicijalizacija podsustava nad segmentom spremnika počevši od adrese `mem_segm` i veličine `size` – na početku tog segmenta stvara se potrebna

[1]Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_03_Interrupts/03_Dynamic_memory`.

struktura podataka za upravljanje ovom metodom (zapravo zaglavlje liste slobodnih blokova)

- `ffs_alloc(mpool, size)` – zahtjev za blokom spremnika veličine `size` iz skupine blokova upravljanih strukturom `mpool` (engl. *memory pool*)
- `ffs_free(mpool, chunk)` – oslobađanje bloka `chunk` i vraćanje skupu blokova upravljanog strukturom `mpool`.

Na početku svakog bloka u listi slobodnih (struktura `ffs_hdr_t`) nalazi se:

- `size` – veličina bloka (uključujući početno i krajnje zaglavlje)
- `prev` – kazaljka na prethodni slobodni blok u listi (`NULL` ako je prvi)
- `next` – kazaljka na idući slobodni blok u listi (`NULL` ako je zadnji).

Na kraju svakog slobodnog bloka nalazi se njegova veličina (kao zaglavlje).

Blokovi koji su dodijeljeni (zauzeti blokovi) nisu stavljeni ni u kakve liste te im kazaljke `prev` i `next` u zaglavlju nisu potrebne. Zato se na početku i na kraju svakog zauzetog bloka kao zaglavlje nalazi samo veličina bloka. Najmanje značajan bit veličine za slobodne se blokove postavlja u nulu, a za zauzete u jedinicu (ionako se taj bit ne koristi s obzirom na to da je najmanja jedinica u kojoj se blokovi dijele veća – 4 okteta). Oznaka zauzetosti spriječit će pokušaj spajanja slobodnog bloka sa zauzetim, odnosno omogućit da se upravo oslobođeni blok spoji s blokom koji se u spremniku nalazi neposredno prije ili poslije njega, ako je takav blok također slobodan (prvi bit veličine je nula).

Slika 7.4. Primjer slobodnih i zauzetih blokova i njihova zaglavlja

Slika 7.4. prikazuje jednostavan sustav koji se sastoji od dva slobodna i jednog zauzetog bloka. Zauzeti blok ima oznaku zauzetosti (najmanje značajan bit veličine) postavljenu u jedinici, dok slobodni na istom mjestu imaju nulu. Kazaljke `next` i `prev` pokazuju na početke slobodnih blokova (na njihova zaglavlja).

7.2.2. Metoda TLSF

Svi navedeni algoritmi (i sa svim mogućim poboljšanjima) i dalje imaju ukupnu složenost barem jednaku logaritamskoj. Ako to nije dovoljno, onda treba i nauštrb prosječne učinkovitosti pronaći algoritme koji će dodjelu obavljati u konačnom broju koraka, odnosno čija će složenost biti konstantna $O(1)$ da bi se on mogao primijeniti i u sustavima za rad u stvarnom vremenu.

Jedan od takvih algoritama je TLSF (dvorazinsko polje uređenih listi). Kao i kod *dlmalloc* algoritma i TLSF ima više listi za pohranu slobodnih blokova. Međutim, organizacija tih listi je dvodimenzionalna. U svaku listu stavljaju se blokovi određenih veličina, počevši od minimalne V_{min} do V_{max} koji je bar za jedan manji od iduće liste koja sadrži veće blokove. Pažljivim odabirom veličina blokova i broj listi u jednoj razini postignuto je da se složenost traženja slobodnog bloka odgovarajuće veličine svede u $O(1)$ složenost, tj. bez obzira na broj slobodnih blokova odabir će biti napravljen s unaprijed izračunatim maksimalnim brojem instrukcija.

[2]Slika 7.5. je preuzeta iz [TLSF].

Slika 7.5. Primjer rada TLSF algoritma[2]

Slika 7.5. prikazuje jednu moguću organizaciju listi za slobodne blokove. U jednoj razini ima osam različitih listi počevši s onom u koju se smještaju blokovi veličina od 32 do 35 jedinica (jedinica može biti oktet, ali i veća, primjerice KB). U prikazanom primjeru sve liste u prvoj razini osim jedne su prazne. U njoj se nalaze dva bloka, veličine 38 i 36 jedinica. Slično je i s drugom razinom u kojoj je samo šesta lista neprazna. Preostale dvije razine su također prazne.

Radi brže pretrage koriste se i polja bitova, koja se mogu označiti kao matrica $SL[4,8]$ koja imaju jedinice samo na onim mjestima za koje ekvivalentna lista nije prazna (maske). Radi brže pretrage tih polja (da se i to može napraviti u jednom koraku) koristi se i još jedno polje bitova $FL[4]$ (za *first level directory*, prema slici 7.5.).

Postupak pretrage za blokom određene veličine prvo kreće od pronalaska indeksa fl razine koja sadrži listu s najmanjim, a ipak odgovarajućim blokovima (engl. *first level index*). Potom se računa indeks sl liste u toj razini koja sadrži takve blokove (engl. *second level index*). Nakon toga se korištenjem maski i izračunatih indeksa dolazi do prve liste s odgovarajućim slobodnim blokovima.

Primjerice, ako se traži slobodan blok veličine 84 jedinice algoritam će dati brojke $fl = 1$ te $sl = 3$ (ako pretpostavimo da numeracija ide od brojke 0 kao što je to uobičajeno u C-u). Pretraga će dakle započeti od liste $LISTA[1,3]$ koja sadrži blokove veličina od 88 do 95!

Na prvi pogled to može iznenaditi, jer se možda i u prethodnoj listi – $LISTA[1,2]$ koja sadrži blokove velike od 80 do 87 jedinica nalazi dovoljno veliki blok. Međutim, tamo se mogu nalaziti i manji blokovi pa bi opet trebalo slijedno pretraživati što nije $O(1)$ složenost. Zato se u algoritmu kreće od liste koja sigurno sadrži (ako nije prazna) dovoljno velike blokove te algoritam nije *najbolji odgovarajući* (engl. *best–fit*) već samo *prikladan* (engl. *good–fit*).

Korištenjem maski se u dva koraka nalazi neprazna lista s dovoljno velikim slobodnim blokovima. U prvom koraku se pronalazi prva razina $FL[i]$ počevši od prethodno izračunate ($fl = 1$), a u drugom se koraku pretragom bitova $SL[1,j]$ traži prva neprazna lista te razine počevši od izračunate ($sl = 3$). Za zadani primjer pretraga po $SL[1,j]$ će dati indekse $(1,5)$ kao indekse liste iz koje treba uzeti blok. Svi blokovi u toj listi su veći od tražene 82 jedinice, ali i najveći mogući (111) nije veći za 32 jedinice (ili više) pa se on u cijelosti dodjeljuje. U protivnom bi se blok podijelio, a ostatak kao slobodni blok stavio u odgovarajuću listu.

Kada bi zahtjev bio za blokom veličine 50 jedinica, onda bi izračunata startna točka bila $(fl, sl) = (0,5)$. Međutim, korištenjem maski ustanovilo bi se da u prvoj razini nema liste s odgovarajućim slobodnim blokovima te bi se pretraga ponovila s indeksima prve liste iduće razine, tj. s $(fl, sl) = (1,0)$. Pretraga po maskama bi konačno pronašla opet istu listu $(104-111)$,

tj. $LISTA[1,5]$. Sada će blok biti zaista prevelik i podijelit će se na dva dijela. Pretpostavimo da je blok koji se iz te liste oduzeo bio velik 108 jedinica. Uz to će ostatak biti 108-50=58 jedinica (zanemari li se veličina zaglavlja u ovom jednostavnom proračunu!). Slobodan blok te veličine smješta se u listu $LISTA[0,6]$ te se u bitovima maski prve razine postavlja jedinica za tu listu (na mjestu $SL[0,6]$).

Po istom načelu ostvaren je i algoritam dinamičkog upravljanja spremnikom. On je zasebno ostvaren u okviru Benua, kôd nije preuzet s [TLSF] te se on interno naziva *GMA* (od engl. *grid memory allocation*). Kao i u prikazanome primjeru, najmanji blokovi kreću se od 32 okteta, ali svaka razina ima 32 liste te je i raspodjela po razini u pogledu veličina malo drukčija. Zaglavlja blokova na sličan su način definirana kao u *dlmalloc* rješenju, a detaljnije su opisana u izvornim kôdovima `include/lib/gma.h` i `lib/mm/gma.c`. Sučelje za rad s *GMA* načinom upravljanja spremnikom slično je već opisanom (`ff_simple`).

7.3. Sučelje jezgre za korištenje gomile

S obzirom na to da su u kôdu ostvarena dva načina dinamičkog upravljanja spremnikom, odabir onog koji će se koristiti obavlja se makroom `MEM_ALLOCATOR` u datoteci `Makefile`. U `kernel/memory.h` u ovisnosti o vrijednosti makroa koristi se jedna ili druga metoda. Sučelje koje se u jezgri koristi za rad s dinamičkim spremnikom je:

- `k_mem_init(segment, size)` – za inicijalizaciju
- `kmalloc(size)` – za zahtjeve za blokovima
- `kfree(addr)` za vraćanje blokova u skupinu slobodnih.

Gomila se sastoji od segmenta spremnika koji se nalazi iza jezgre i programa, početnom adresom definiranom u skripti povezivača pa sve do kraja spremničkog prostora (veličina spremnika zadana je konfiguracijskoj datoteci). Navedena inicijalizacija ostvarena je dijelom u `kernel/memory.c` i dijelom u `arch/i386/memory.c`.

7.4. Liste

Mnogi podsustavi jezgre imaju potrebe za organiziranom strukturom podataka. Jedna takva organizacija su liste. S obzirom na to da liste trebaju razni sustavi, osnove operacije za rad s listama su izdvojene, kao zasebna cjelina, koju mogu svi koristiti, uz uključivanje potrebne strukture podataka i korištenje ostvarena sučelja. U četvrtoj se fazi inkrementa liste koriste za registraciju prekida.

Lista, odnosno strukture podataka i operacije nad njima definirane su u zaglavlju `include/lib/list.h` i ostvarene u `lib/list.c`.

Isječak kôda 7.1. Chapter_03_Interrupts/03_Dynamic_memory/include/lib/list.h

```
43 /*! List element pointers */
44 typedef struct _list_h_
45 {
46         struct _list_h_  *prev;
47                           /* pointer to previous list element */
48
49         struct _list_h_  *next;
50                           /* pointer to next list element */
51
52         void             *object;
53                           /* pointer to object (which contains this list_h) */
54 }
55 list_h;
```

Uobičajeno, tu su kazaljke koje povezuju susjedne objekte u listi (`prev`, `next`). Ipak te kazaljke ne pokazuju na početak objekta već na zaglavlje koje se koristi za liste koje se nalazi negdje unutar objekta! Zato je i potrebna kazaljka na početak objekta (`object`). Teoretski bi bilo to moguće napraviti i bez ove dodatne kazaljke, ali bismo onda bili ograničeni na samo jednu listu po objektu, a navedeno zaglavlje (kraće za `void *object`) moralo bi biti na početku objekta.

Podatkovna struktura koja želi koristiti navedenu (dvostruko povezanu) listu, u svaki svoj element mora dodati strukturu `list_h`.

Struktura `list_h` mogla bi biti i izvan objekta za koji se koristi i svaki put zauzimati prilikom dodavanja novog objekta u listu te oslobađati prilikom otpuštanja. Iako ovaj mehanizma izgleda jednostavniji i prilagodljiviji, odabrano je načelo ugradnje u sam objekt s obzirom na to da su objekti gotovo uvijek u listama, a samom ugradnjom se izbjegava dodatno trošenje i vremena i spremnika: "vremena" zato što bi pri svakom dodavanju novog elementa trebalo dohvatiti spremnički prostor za jedan `list_h` objekt (slično pri micanju treba taj objekt vratiti), "spremnika" zato što se pri zauzimanju novog `list_h` objekta dodatno zauzimaju i zaglavlja za taj blok, a ta su zaglavlja sumjerljiva s `list_h`.

Zaglavlje liste je tipa `list_t`, a sadrži kazaljke na prvi i zadnji element liste, odnosno pokazuju na `list_h` elemente u tim objektima.

Isječak kôda 7.2. Chapter_03_Interrupts/03_Dynamic_memory/include/lib/list.h

```
57 /*! list header type */
58 typedef struct _list_
59 {
60         list_h  *first;
61         list_h  *last;
62 }
63 list_t;
```

Slika 7.6. prikazuje jedan primjer ostvaren navedenim strukturama podataka i pravilima povezivanja.

Slika 7.6. Primjer liste

Operacije nad listama ostvarene su u `lib/list.c`. U nastavku je radi ilustracije prikazana operacija dodavanja elementa na kraj liste.

Isječak kôda 7.3. Chapter_03_Interrupts/03_Dynamic_memory/lib/list.c

```
23 /*! Add element to list, add to tail - as last element */
24 void list_append(list_t *list, void *object, list_h *hdr)
25 {
26         ASSERT(list && object && hdr);
27
28         hdr->object = object; /* save reference to object */
```

```
29          hdr->next = NULL; /* put it at list end (as last element) */
30
31          if (list->first)
32          {
33                  list->last->next = hdr;
34                  hdr->prev = list->last;
35                  list->last = hdr;
36          }
37          else {
38                  list->first = list->last = hdr;
39                  hdr->prev = NULL;
40          }
41  }
```

7.5. Registracija više funkcija za obradu istog prekida

Dodavanjem dinamičkog upravljanja spremnikom i listi proširen je i prekidni podsustav[3] mogućnošću registracije više funkcija na isti prekid. Svaka registracija stvara objekt i dodaje ga u listu za zadani prekid.

Isječak kôda 7.4. Chapter_03_Interrupts/04_Interrupts/arch/i386/interrupt.c

```
15  /*! interrupt handlers */
16  static list_t ihandlers[INTERRUPTS];
17
18  struct ihndlr
19  {
20          int (*ihandler)(unsigned int);
21
22          list_h list;
23  };
```

Za svaki je prekid definirana jedna lista, tj. jedno zaglavlje. U listu će se stavljati registracije prekida u obliku objekta `struct ihndlr`.

Isječak kôda 7.5. Chapter_03_Interrupts/04_Interrupts/arch/i386/interrupt.c

```
50  void arch_register_interrupt_handler(unsigned int inum, void *handler)
51  {
52          struct ihndlr *ih;
53
54          if (inum < INTERRUPTS)
55          {
56                  ih = kmalloc(sizeof(struct ihndlr));
57                  ASSERT(ih);
58
59                  ih->ihandler = handler;
60
61                  list_append(&ihandlers[inum], ih, &ih->list);
62          }
63          else {
64                  LOG(ERROR, "Interrupt %d can't be used!\n", inum);
65                  halt();
66          }
67  }
```

Pri registraciji nove funkcije za obradu prekida stvara se novi "opisnik" za tu registraciju i dodaje se u listu. Slično, pri brisanju registracije se taj element miče iz liste.

[3] Proširenje je napravljeno u fazi `Chapter_03_Interrupts/04_Interrupts`.

Isječak kôda 7.6. Chapter_03_Interrupts/04_Interrupts/arch/i386/interrupt.c

```
93  void arch_interrupt_handler(int irq_num)
94  {
95          struct ihndlr *ih;
96
97          if (irq_num < INTERRUPTS && (ih = list_get(&ihandlers[irq_num], FIRST)))
98          {
99                  /* enable interrupts on PIC immediately since program may not
100                  * return here immediately */
101                 if (icdev->at_exit)
102                         icdev->at_exit(irq_num);
103
104                 /* Call registered handlers */
105                 while (ih)
106                 {
107                         ih->ihandler(irq_num);
108
109                         ih = list_get_next(&ih->list);
110                 }
111         }
```

Pri pojavi nekog prekida prolazi se kroz listu registriranih funkcija za taj prekid te se one sve pozivaju redom kojim su navedene u listi – redom prijave u prekidni podsustav.

Pitanja za vježbu 7

1. Opisati postupke statičkog i dinamičkog upravljanja spremnikom. Koje su prednosti, a koji nedostaci pojedinih postupaka?
2. Koja je složenost algoritama: *prvi odgovarajući*, *najbolji odgovarajući*, *dlmalloc*, *Buddy* te TLSF?
3. Što je to fragmentacija (kod dinamičkog upravljanja spremnikom)? Koji algoritmi imaju veću, a koji manju fragmentaciju? Što je to unutarnja fragmantacija (primjerice kod *Buddy* algoritma)?
4. Koje osnovno sučelje treba nuditi podsustav za dinamičko upravljanje spremnikom?
5. Navesti operacije nad listom te njihovu složenost.

8. Upravljanje vremenom

Upravljanje vremenom je od kritičnog značaja za ugrađene sustave, a pogotovo za sustave za rad u stvarnom vremenu. Ostali sustavi također trebaju taj podsustav kako za upravljanje sklopovskim tako i za programske komponente. Razni sklopovski elementi zahtijevaju periodičku provjeru i ažuriranja ili obrade. U sustavima gdje se raspoređivanje zadataka (dretvi/procesa) obavlja podjelom vremena bitno je voditi evidenciju o korištenju procesorskog vremena od strane svakog pojedinog zadatka. Tada se zapravo govori o virtualnom vremenu, ali i za to je potreban sat koji se može očitati te pomoću njega izračunati utrošeno vrijeme.

Nadalje, i sami zadaci trebaju sučelje koje će im omogućiti uvid u trenutno vrijeme, tražiti odgodu svog izvođenja ili dobivanje periodičkih signala za pokretanje akcija.

Zahtjevi prema podsustavu za upravljanje vremenom dolaze od:

- jezgre – potrebe za upravljanjem podsustavima (primjerice raspoređivanje), i
- zadataka – očitanje trenutnog vremena, odgoda izvođenja, periodički signali.

Zahtjevi se mogu podijeliti u nekoliko skupina:

- očitanje trenutnog vremena sata
- programiranje jedne *akcije* u zadanom budućem trenutku
- programiranje *periodičke akcije* (periodički se javlja)
- odgode dretve do zadanog budućeg trenutka (ili za zadani interval).

Akcija (obična ili periodička) može uključivati:

- pozivanje određene funkcije (sa zadanim parametrima)
- slanje signala zadanoj dretvi te
- propuštanje zaustavljene dretve ("buđenje" odgođene dretve).

Sučelje kojim se mogu ostvariti navedeni zahtjevi i akcije može se načelno opisati funkcijom:

```
postavi_alarm (
    kada,
    period,
    funkcija_aktivacije,
    parametar_funkcije,
    zaustaviti_dretvu
);
```

Korišten je pojam *alarm* jer sugerira da se po njegovoj aktivaciji treba nešto hitno napraviti, što ovdje i jest slučaj[1]. Kratki opis parametara slijedi u nastavku.

- Parametar `kada` definira neki budući trenutak kada se alarm treba (prvi put) aktivirati.
- Parametar `period` definira period za periodičke alarme. Periodički alarm se prvi put aktivira u `kada` a nakon toga svakih `period` jedinica vremena. Ako nije zadan (`period` je nula), alarm nije periodički.
- Parametri `funkcija_aktivacije` i `parametar_funkcije` definiraju akciju koju treba pokrenuti u trenutku aktivacije alarma. Ako `funkcija_aktivacije` nije zadana (jednaka je `NULL`) tada se neće ništa pokrenuti osim možda omogućiti nastavak rada zadanoj dretvi, ako je tako zadano zadnjim parametrom.

[1] Izvorni engleski termin za *alarm* je *timer*.

- Parametar `zaustaviti_dretvu` definira treba li zaustaviti dretvu koja poziva navedenu funkciju do aktivacije alarma. Naime, za operacije *odgodi* potrebno je zaustaviti dretve. Više o mehanizmu odgode dretvi u odjeljku 11.3.3.

Prije opisa načina upravljanja vremenom slijedi kratki opis sklopovlja koje se koristi za upravljanje vremenom.

8.1. Korištenje sklopa *Intel 8253*

Način rada većine sklopovlja za upravljanje vremenom je jednostavan: sastoje se od brojila koje određenom stalnom frekvencijom odbrojava od zadane vrijednosti (učitane početno u brojilo) do nule (ili obrnuto). Kada dođe do nule može izazvati prekid (ako se tako programira), ponovno učitava početnu vrijednost i opet odbrojava prema nuli. Osim početne vrijednosti nekim se sklopovima može odrediti i djelilo ulazne frekvencije (kada su potrebna sporija otkucavanja).

Iako modernija računala imaju i druge (bolje) mogućnosti za upravljanje vremenom, ovdje se koristi sklop *Intel 8253* koji spada u skupinu programirljivih brojila (engl. *programmable interval timers – PIT*) koji je prisutan u svim sustavima i jednostavan je za upravljanje.

Sklop *Intel 8253* ima tri brojila, ali se najčešće za potrebe upravljanja vremenom koristi samo jedno – prvo. Ostala brojila su u povijesti imala i druge namjene: 2. za osvježavanje DRAM-a, 3. za generiranje signala za interni zvučnik, tzv. *speaker*. Osnovna frekvencija s kojom sklop radi je 1193181.8 Hz i odabrana je iz povijesnih razloga – to je trećina frekvencije signala za televizor prema normi *NTSC*. Navedena frekvencija omogućuje, uz korištenje raznih vrijednosti brojila, najveću frekvenciju generiranja prekida od nešto preko jednog MHz (svaki otkucaj je prekid) do najmanje od 18.2 Hz (kad se koristi najveća početna vrijednost brojila).

Sklop ima i druge mogućnosti (generiranje električnog signala), ali trenutno se od njega koristi samo mogućnost brojila i generiranja prekida. Više detalja o sklopu, načinu programiranja i slično može se naći u literaturi [Intel 8253].

Korištenja sklopa ostvaruje se kroz sloj *arch*, pomoćnim funkcijama ostvarenim u datotekama `arch/i386/drivers/i8253.c`[2]. S obzirom na to da se umjesto njega mogu koristiti i bolja brojila novijih procesora, sklop se koristi preko sučelja `arch_timer_t` (definiranim u `arch/i386/time.h`) tako da se on može jednostavnije zamijeniti.

Osnovne operacije uključuju čitanje trenutne vrijednost brojila te unos nove početne vrijednosti za odbrojavanje. Nadalje, ovdje se nalazi i sučelje koje dozvoljava ili maskira prekid sklopa korištenjem sučelja prema sklopu *Intel 8259* (ulaz IR0). Također, tu su ostvarene i pomoćne funkcije za pretvorbu broja iz brojila u vrijeme i obratno, potrebne konstante i makroi.

Korištenjem sklopa *Intel 8253* preko `arch_timer_t` sučelja, u sloju *arch* u datoteci `arch/i386/time.c` ostvaren je osnovni podsustav za upravljanje vremenom. Temeljna funkcionalnost koju on pruža jezgri je u postavljanju alarma:

```
arch_timer_set ( kada, funkcija_aktivacije );
```

po čijem se isteku poziva zadana funkcija jezgre. U sloju jezgre treba ostvariti strukturu podataka za praćenje i upravljanje svim alarmima (kada i kojim ih redoslijedom aktivirati).

8.2. Osnovni podsustav za upravljanje vremenom

Osnove podsustava za upravljanje vremenom ostvarene su u sloju *arch* koji ionako upravlja brojilima nad kojima se sustav ostvaruje.

[2] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_04_Timer`.

Sklop koji se (i općenito) koristi za upravljanje vremenom može se idejno opisati sa sljedećim elementima i akcijama:

- registar `zadnje_učitano`
  - registar koji pamti zadnju poslanu vrijednost u registar `brojilo`, od koje treba početi brojati prema nuli.
- registar `brojilo`
  - registar koji na svaki signal oscilatora smanjuje vrijednost za jedan;
  - kada mu vrijednost dođe do nule izaziva se zahtjev za prekid, učitava vrijednost iz registra `zadnje_učitano` te ponavlja brojanje (prema nuli);
  - čitanjem ovog registra dobiva se trenutna vrijednost brojila te se može izračunati protok vremena;
  - upisivanjem u ovaj registar postavlja se nova početna vrijednost koja se i zapamti u registru `zadnje_učitano`;

Slika 8.1. prikazuje odbrojavanje brojila od zadane vrijednosti (zv) do nekog budućeg trenutka ili do nule kada izaziva prekid.

Slika 8.1. Korištenje brojila za mjerenje protoka vremena i izazivanje prekida

Upravljanje vremenom korištenjem opisanog brojila može se pojednostavljeno prikazati sljedećim pseudokodom:

```
postavi_alarm ( odgoda, akcija ) /* arch sloj */
{
    sat += BROJ_U_VRIJEME ( zadnje_učitano - brojilo );

    preostala_odgoda = odgoda;
    zadnje_učitano = VRIJEME_U_BROJ ( preostala_odgoda );
    ako ( zadnje_učitano > BR_MAX )
        zadnje_učitano = BR_MAX;

    brojilo = zadnje_učitano;
    funkcija_aktivacije = akcija;
}

prekid_brojila () /* prekid se zbiva kad brojilo dođe do nule */
{
    sat += BROJ_U_VRIJEME ( zadnje_učitano );
    ako ( funkcija_aktivacije == NULL ) // ili preostala_odgoda <= 0 (*)
    {
        zadnje_učitano = BR_MAX;
```

```
        brojilo = zadnje_učitano;
    }
    inače {
        preostala_odgoda -= BROJ_U_VRIJEME ( zadnje_učitano );
        zadnje_učitano = VRIJEME_U_BROJ ( preostala_odgoda );

        ako ( zadnje_učitano > BR_MAX || zadnje_učitano <= 0 )
            zadnje_učitano = BR_MAX;

        brojilo = zadnje_učitano;

        ako ( preostala_odgoda <= 0 ) {
            akcija = funkcija_aktivacije;
            funkcija_aktivacije = NULL;
            akcija (); /* ovdje se opet može postavljati alarm! */
        } // uz (*) može samo: funkcija_aktivacije ();
    }
}
```

Osim već spomenutih registara brojila, u kodu se koriste i varijable:

- `sat` – trenutna vrijednost sata sustava (koja se izračunava), varijabla `clock` u sloju *arch*
- `zadnje_učitano` – zadnja poslana vrijednost u `brojilo`, varijabla `last_load` u sloju *arch*
- `preostala_odgoda` – koliko još do iduće aktivacije alarma, varijabla `delay` u sloju *arch*.

U sloju *arch* nalaze se ostvarenja gornjeg pseudokoda. Npr. svaki put kada se dogodi prekid sklopa s brojilom poziva se `arch_timer_handler`:

Isječak kôda 8.1. Chapter_04_Timer/03_Timers/arch/i386/time.c

```
119 static void arch_timer_handler()
120 {
121         void (*k_handler)();
122
123         time_add(&clock, &last_load);
124
125         if (alarm_handler)
126         {
127                 time_sub(&delay, &last_load);
128
129                 if (time_cmp(&delay, &threshold) <= 0)
130                 {
131                         /* activate alarm; but first update counter */
132                         last_load = timer->max_interval;
133                         timer->set_interval(&last_load);
134
135                         k_handler = alarm_handler;
136                         alarm_handler = NULL; /* reset kernel callback function */
137                         k_handler(); /* forward interrupt to kernel */
138                 }
139                 else {
140                         if (time_cmp(&delay, &timer->min_interval) < 0)
141                                 last_load = timer->min_interval;
142                         else if (time_cmp(&delay, &timer->max_interval) < 0)
143                                 last_load = delay;
144                         else
145                                 last_load = timer->max_interval;
146
147                         timer->set_interval(&last_load);
148                 }
149         }
150 }
```

U funkciji se najprije ažurira sat sustava. Potom se razmatra alarm. Ako je proteklo alarmom zadano vrijeme on se aktivira pozivanjem zadane jezgrene funkcije (jezgra postavlja alarm i zadaje svoju funkciju aktivacije). Prije prosljeđivanja poziva jezgri, alarm se postavlja na početnu vrijednost (*resetira*). Ako jezgra treba novi alarm ona će ga sama postaviti.

Sloj *arch* nudi mogućnost samo jednog alarma. Sve složenije operacije treba ostvariti u sloju jezgre (više alarma, periodičke alarme i slično).

Upravljanje vremenom ostvareno je kroz nekoliko faza. Prvo je dodano samo sučelje za dohvaćanje i postavljanje sata sustava. U idućoj fazi je dodano sučelje za samo jedan alarm. Tek je u sljedećoj fazi dodano puno sučelje za rad s alarmima (ne računajući proširenja dodana dodavanjem višedretvenosti).

Prije opisa ostvarenja upravljanja vremenom u Benu (u jezgri), slijedi opis POSIX sučelja prema programima (dretvama) koje je korišteno u ostvarenju.

8.3. POSIX sučelje za upravljanje vremenom

POSIX sučelje [POSIX] pretpostavlja mogućnost postojanja više satnih mehanizama. Gotovo sve funkcije za upravljanje vremena primaju identifikator sata kojeg trebaju koristiti. Dva osnovna sata su:

- `CLOCK_REALTIME` – sat sustava koji odbrojava sukladno stvarnom protoku vremena, ali se može podešavati posebnim sučeljem te
- `CLOCK_MONOTONIC` – sat koji odbrojava sukladno stvarnom protoku vremena, ali se ne može mijenjati.

Drugi sat, `CLOCK_MONOTONIC`, može služiti za primjene kod kojih nije potrebno znati točno trenutno vrijeme već se samo koriste intervali. Uvedeno je zbog mogućih problema koje može nastati korištenjem sata `CLOCK_REALTIME` i njegovom promjenom (primjerice usklađivanjem s udaljenim poslužiteljem), tj. kako će ta promjena utjecati na već postavljene alarme. Benu trenutno ostvaruje samo `CLOCK_REALTIME`.

POSIX sučelje za upravljanje vremenom možemo podijeliti na funkcije za upravljanje satom i funkcije za upravljanje alarmima. Sučelja su definirana u `include/api/time.h` za programe te u `kernel/time.h` za interne potrebe jezgre.

8.3.1. Upravljanje satom

U funkcije za upravljanje satom spadaju funkcije za dohvat i postavljanje trenutnog vremena.

Isječak kôda 8.2. Chapter_04_Timer/03_Timers/include/api/time.h

```
7  int clock_gettime(clockid_t clockid, timespec_t *time);
8  int clock_settime(clockid_t clockid, timespec_t *time);
```

Prvi parametar funkcija `clockid` je identifikator sata, a drugi kazaljka na vrijeme koje treba postaviti (za `clock_settime`), odnosno kamo ga treba pohraniti (za `clock_gettime`). Vrijeme je definirano sekundama i nanosekundama u strukturi `timespec_t`.

8.3.2. Upravljanje alarmima

POSIX sučelje za upravljanje alarmima donekle prati već opisano ponašanje alarma. Osnovno načelo s alarmima sastoji se u *stvaranju* alarma, kada se definira što se treba dogoditi pri aktivaciji alarma te *postavljanju* alarma, kada se postavlja vrijeme kad alarm treba aktivirati (tada je alarm aktivan, engl. *armed*).

Isječak kôda 8.3. Chapter_04_Timer/03_Timers/include/api/time.h

```
13 int timer_create(clockid_t clockid, sigevent_t *evp, timer_t *timer);
```

Pri stvaranju alarma (`timer_create`) preko strukture `sigevent_t` definira se akcija koju treba poduzeti pri aktiviranju alarma (kada zadano vrijeme istekne).

Struktura `sigevent_t` koristi se i općenitije za definiranje akcije na neki događaj, ne samo za alarme već i za *signale*[3]. Struktura `sigevent_t` je definirana u `include/types/signal.h` i sastoji se od:

- `sigev_notify` – definira način akcije na događaj:
  - `SIGEV_NONE` – nema akcije
  - `SIGEV_SIGNAL` – akcija na događaj je slanje signala `sigev_signo`
  - `SIGEV_THREAD` – akcija na događaj je stvaranje nove dretve koja obrađuje događaj funkcijom `sigev_notify_function` uz parametar `sigev_value`
  - `SIGEV_THREAD_ID` – signal se ne šalje pozivajućoj dretvi (koja je postavila alarm) već dretvi definiranoj sa `sigev_notify_thread_id` (proširenje koje donosi Linux, a nije definirano POSIX-om)
- `sigev_signo` – identifikacijski broj signala koji se šalje (ako se signal šalje kao aktivacija događaja)
- `sigev_value` – vrijednost (broj ili kazaljka) koja se šalje uz aktivaciju događaja uz signal ili kao parametar funkcije `sigev_notify_function`
- `sigev_notify_function` – početna funkcija nove dretve stvorene kao reakcija na aktiviranje događaja (uz `sigev_notify==SIGEV_THREAD`)
- `sigev_notify_attributes` – postavke za novu dretvu
- `sigev_notify_thread_id` – opisnik dretve kojoj treba poslati signal (ako je to akcija na događaj aktivacije).

Opisnik stvorenog alarma (u funkciji `timer_create`) sprema se na adresu zadanu u varijablom `timer`. Ista se adresa koristi u idućim funkcijama (nakon stvaranja alarma).

Isječak kôda 8.4. Chapter_04_Timer/03_Timers/include/api/time.h

```
14 int timer_delete(timer_t *timer);
```

Funkcija `timer_delete` briše stvoreni alarm – miče ga iz sustava.

Postavljanje i brisanje vremena aktivacije obavlja se s `timer_settime`.

Isječak kôda 8.5. Chapter_04_Timer/03_Timers/include/api/time.h

```
15 int timer_settime(timer_t *timer, int flags, itimerspec_t *value,
16                   itimerspec_t *ovalue);
```

Zastavica `flags` može imati postavljenu zastavicu `TIMER_ABSTIME` kojom označava da je zadano vrijeme u `value` apsolutno. Ako zastavica nije postavljena onda se vrijeme u `value` smatra relativnim u odnosu na trenutno vrijeme ("aktiviraj alarm za N jedinica vremena", a ne "aktiviraj alarm kada sat bude jednak N" kako se interpretira ako je zastavica `TIMER_ABSTIME` postavljena).

Vrijeme je zadano varijablom `value` koja sadrži *vrijeme prve aktivacije* (`it_value` dio strukture `itimerspec_t`) te *period* (`it_interval` dio strukture `itimerspec_t`). Ako je period jednak nuli onda alarm nije periodički. Ako je vrijeme prve aktivacije jednako nuli onda se

[3]Opis mehanizma signala dan je kasnije u odjeljku 11.5.

alarm deaktivira (ne i briše!). Ako vrijeme prve aktivacije nije jednako nuli alarm se postavlja u aktivno stanje (odbrojava) te će nakon zadanog vremena (ili u zadano vrijeme uz zastavicu `TIMER_ABSTIME`) biti aktiviran.

Zadnji parametar funkcije `timer_settime` omogućava pohranu trenutnog stanja alarma – dohvaća se vrijeme do iduće aktivacije i pohranjuje na zadanu adresu.

Isječak kôda 8.6. Chapter_04_Timer/03_Timers/include/api/time.h

```
17 int timer_gettime(timer_t *timer, itimerspec_t *value);
```

Funkcija `timer_gettime` vraća vrijeme do iduće aktivacije zadanog alarma.

8.3.3. Odgoda izvođenja programa

POSIX sučelje za odgodu definirano je funkcijama:

Isječak kôda 8.7. Chapter_04_Timer/03_Timers/include/api/time.h

```
 9 int clock_nanosleep(clockid_t clockid, int flags, timespec_t *request,
10                     timespec_t *remain);
11 int nanosleep(timespec_t *request, timespec_t *remain);
```

Druga je funkcija skraćena inačica prve uz `CLOCK_REALTIME` kao prvi parametar te nula kao drugi parametar (`flags`).

Parametar `request` definira vrijeme odgode, "koliko dugo" treba odgoditi izvođenje programa kada je `flags=0`, odnosno "do kada" uz `flags=TIMER_ABSTIME`.

U intervalu od početka odgode (trenutak poziva gornjih funkcija) do trenutka kada je odgoda trebala završiti može se mnogo toga dogoditi. Neki od tih događaja mogu uzrokovati i prekid odgode. Ako se tako nešto dogodilo i odgoda je trajala kraće od zadanog vremena, onda se na adresu `remain` (ako nije `NULL`) pohranjuje vrijednost intervala vremena od trenutka prekida odgode do trenutka kada je odgoda trebala završiti (dio odgode koja je preostala – "neprospavano vrijeme").

8.4. Upravljanje vremenom ostvareno u jezgri

8.4.1. Pozivi jezgrenih funkcija

POSIX sučelja prikazana u prethodnom odjeljku definirana su u sloju *api*, ali su ostvarena u sloju *jezgre*. S obzirom na to da navedene operacije trebaju (ili mogu trebati) i samoj jezgri za interne potrebe, one se iz programa pozivaju preko zasebnog sučelja. Primjerice, stvaranje alarma se iz programa poziva s `timer_create`, iz sloja *api* koriste se pozivi jezgre `sys__timer_create`, a u toj jezgrenoj funkciji poziva se `ktimer_create`. Kada jezgra treba stvoriti alarm za svoje potrebe ona koristi samo zadnju funkciju (`ktimer_create`). Navedeno uslojavanje prikazuje slika 8.2.

sloj	funkcija koja se poziva iz sloja
programs	timer_create
api	sys__timer_create
kernel	ktimer_create

Slika 8.2. Pozivi između slojeva na primjeru `timer_create`

Jedan od razloga uslojavanja jest u odvajanju jezgre i programa. Bitniji razlog jest u stanju sustava: kada se funkcija poziva iz programa sustav se nalazi u tom (*korisničkom*) načinu rada

(primjerice prekidi su dozvoljeni) dok kad se funkcija poziva iz jezgre stanje sustava je drukčije (primjerice prekidi su zabranjeni). Izgled tih funkcija "omotača", tj. funkcija koje započinju sa `sys__` može se vidjeti na jednoj kratkoj funkciji.

Isječak kôda 8.8. Chapter_04_Timer/03_Timers/kernel/time.c

```
397 int sys__clock_gettime(clockid_t clockid, timespec_t *time)
398 {
399         int retval;
400
401         SYS_ENTRY();
402
403         ASSERT_ERRNO_AND_EXIT(
404                 time && (clockid==CLOCK_REALTIME || clockid==CLOCK_MONOTONIC),
405                 EINVAL
406         );
407
408         retval = kclock_gettime(clockid, time);
409
410         SYS_EXIT(retval, retval);
411 }
```

Makro `SYS_ENTRY()` zabranjuje daljnje prekidanje, ali i pohranjuje prethodno stanje (jesu li ili nisu i prije bili zabranjeni prekidi). Makro `SYS_EXIT(ENUM,RETVAL)` vraća prethodno stanje (omogućava prekidanje ako je bilo omogućeno prije poziva te funkcije) te pritom postavlja *oznaku greške* (engl. *error number*) na `ENUM` te vraća vrijednost `RETVAL` kao povratnu vrijednost jezgrene funkcije. Njoj sličan makro `SYS_RETURN(RETVAL)` radi isto, ali ne postavlja oznaku greške (ne mijenja ju).

Većina jezgrenih funkcija kao *povratnu vrijednost* vraća status obavljanja operacije. Ako je operacija uspješno obavljena, osim što će se u oznaku greške postaviti nula, vratit će se nula i kao povratna vrijednost. Ako operacija nije uspješno obavljena, postavlja se potrebna vrijednost u oznaku greške te se vraća vrijednost –1 (konstanta `EXIT_FAILURE`). Funkcije koje su izuzetak, koje koriste povratnu vrijednost za nešto drugo (primjerice vraćanje dijela rezultata operacije), moraju pripaziti na postavljanje oznake greške i načina povratka vrijednosti (mogu to napraviti s `ASSERT_ERRNO_AND_EXIT(EXPR,ENUM)`, `SYS_EXIT` ili kombinacijom makroa `SET_ERRNO(ENUM)` i `SYS_RETURN`).

Oznaka greške ili *identifikator greške* predstavlja status zadnje (jezgrene) funkcije, a govori o uspješnosti obavljanja te funkcije. Kada funkcija uspješno (predviđeno) obavi zahtijevanu operaciju onda je oznaka greške jednaka nuli (konstanta `EXIT_SUCCESS`). Kada se dogodi neka greška, primjerice greška u nekom ulaznom parametru funkcije, tada se u oznaku greške postavlja odgovarajuća vrijednost (primjerice `EINVAL`). Po povratku iz funkcije ispitivanjem njene *povratne vrijednosti* i vrijednosti oznake greške može se utvrditi je li se dogodila greška te ako jest, koja je greška u pitanju – prema oznaci greške.

Makro `ASSERT_ERRNO_AND_EXIT(EXPR,ENUM)` služi za provjeru ulaznih parametara te povratak iz jezgrene funkcije ako oni nisu ispravni, uz postavljanje zadane oznake greške. Uvjete koje ulazni parametri trebaju zadovoljiti treba izraziti s logičkim izrazima u `EXPR`.

8.4.2. Upravljanje alarmima

U trenutnom ostvarenju koristi se samo jedan satni mehanizam: `CLOCK_REALTIME`. Pokušaj korištenja drugih završava greškom (i kao povratnom vrijednošću funkcije i kao broja `EINVAL` u oznaci greške).

Svi aktivni alarmi – alarmi koji imaju postavljeno vrijeme koje još nije isteklo, nalaze se u jednoj listi složenoj prema vremenima aktiviranja – prvi u listi će se prvi aktivirati (ima najbliže vrijeme aktivacije). Vremena aktivacije se interno zapisuje (u opisniku alarma koji jezgra stvara

i koristi) u apsolutnim iznosima – ne relativno u odnosu na trenutnu vrijednost sata.

Primjerice, vrijeme aktivacije prvog alarma u listi može biti postavljeno na 1234 sekunde i 123456789 nanosekundi. Navedena se vrijednost uspoređuje s trenutnim satom. Kada sat ima manju vrijednost, primjerice 1111 sekundi i 111111111 nanosekundi, alarm se neće aktivirati. Kada je vrijednost sata veća, primjerice 1234 sekunde i 200000000 nanosekundi, tada se alarm aktivira u idućoj provjeri liste aktivnih alarma.

Na prekid sata koji se prosljeđuje iz sloja *arch* u jezgru uspoređuje se vrijednost sata i prvog alarma u listi. Ispitivanje se obavlja i nakon ubacivanja novog alarma u sustav kao i nakon micanja nekih (radi ažuriranja brojila).

Način aktiviranja alarma ovisi o strukturi `sigevent_t` koja je predana pri stvaranju alarma. Trenutno (u sustavu bez višedretvenosti i podrške za signale) ostvarena je jedino opcija izravnog pozivanja zadane funkcije (kada je postavljani način akcije `SIGEV_THREAD`). S obzirom na to da se alarmi koriste i za odgodu programa, interno (u jezgri) je dodana još jedna akcija – `SIGEV_WAKE_THREAD`) – koja je trenutno identična prethodnoj – poziva se zadana funkcija (zadana u `sigev_notify_function`). U ovom je slučaju to jezgrena funkcija za propuštanje zaustavljenog programa – funkcija `kclock_wake_up`.

Nakon prethodnog opisa osnovnog načela upravljanja alarmima i satnim mehanizmom, detaljnije o svakoj operaciji može se doznati uvidom u izvorni kôd (datoteka `kernel/time.c`). U nastavku su dodatno objašnjene operacije odgode i prekidanje odgode.

Funkcija `sys__clock_nanosleep` ostvaruje odgodu izvođenja programa (kasnije i dretvi). Odgoda se ostvaruje stvaranjem alarma koji će po svojoj aktivaciji prekinuti *radno čekanje*[4] koje se nalazi u `sys__clock_nanosleep`.

Isječak kôda 8.9. Chapter_04_Timer/03_Timers/kernel/time.c

```
483         do {
484                 enable_interrupts();
485                 suspend();              /* suspend till next interrupt */
486                 disable_interrupts();
487         }
488         while (wake_up == FALSE);
```

Radno čekanje se prekida promjenom varijable (`wake_up`) koje se zbiva u funkciji aktivacije takvog posebnog alarma (funkcija `kclock_wake_up`). Budući da se promjena varijable kojom se prekida radno čekanje zbiva u funkciji koja se poziva iz obrade prekida (asinkrono), varijablu treba označiti kao `volatile`, a da prevoditelj ne bi varijablu smjestio u registar procesora (tada radno čekanje ne bi nikada bilo prekinuto). Unutar petlje radnog čekanja dozvoljava se prekidanje tako da brojilo može izazvati prekid i pozvati odgovarajuće funkcije. Ako procesor ima instrukciju koja može zaustaviti procesor do pojave prekida, onda ju se može postaviti u makro `suspend` (kao što je instrukcija `hlt` kod arhitekture x86). Takva instrukcija može smanjiti zagrijavanje procesora i njegovu potrošnju što je vrlo bitno kod ugrađenih sustava napajanih baterijom.

Kada bi postojao razlog prekidanja odgode onda bi u trenutku prekida odgode trebalo pozvati funkciju `kclock_interrupt_sleep`. Ona izračunava i zapisuje dio odgode koji nije ostvaren (ako je to traženo u funkciji odgode). Također se briše i interni alarm koji je trebao prekinuti odgodu. Primjer razloga prekida odgode jest prekid naprave koja traži akciju od programa (ne samo obradu prekida). Naprave su sustavno dodane tek u 5. poglavlju (`Chapter_05_Devices`).

[4] Uvođenjem višedretvenosti radno čekanje se zamjenjuje zaustavljanjem dretve u redu alarma te aktiviranjem druge dretve (prve pripravne).

8.5. Neke mogućnosti drukčijeg upravljanja vremenom

Praćenje sata sustava u mnogim je operacijskim sustavima ostvareno na ponešto drukčiji način. Umjesto korištenja varijabli koje koriste vrijeme u sekundama, češće se koriste brojači koji broje koliko je *osnovnih perioda* (engl. *jiffy*) prošlo od neke početne točke. Pretvorba takvih brojača u vrijeme vrlo je jednostavno: brojač se množi duljinom periode.

Nadalje, alarmi su često uređeni korištenjem relativnog odnosa vremena, a ne apsolutnog kao u prikazanom ostvarenju. Na primjer, prvi alarm u listi aktivnih alarama u varijabli koja se koristi za utvrđivanje vremena njegove aktivacije ima brojač koji se smanjuje za jedan na svaki protok osnovne periode. Kada brojač dođe do nule, alarm se aktivira. Lista alarma je također složena prema vremenima aktivacije, ali svaki sljedeći alarm u listi ima relativno vrijeme buđenja u odnosu na svog prethodnika, a ne apsolutno kao u Benu.

Struktura podataka može biti na razne načine organizirana. U nekoj minimalnoj izvedbi moguće je koristiti i samo jednu vrijednost (brojač) koja se nalazi u opisniku dretve (primjerice `zadano_kašnjenje` kao u [Budin, 2010]).

Osim prikazanog i korištenog sklopa *Intel 8253*, satni podsustav može koristiti i druge brojače i satne mehanizme prisutne u novijim arhitekturama x86 (npr. preko sklopa *ACPI*, brojač na lokalnom sklopu *APIC* te *TSC* registra procesora). Neki od navedenih brojača pružaju znatno više mogućnosti (preciznosti).

U sustavima gdje je potrebna velika vremenska preciznost (primjerice radi upravljanja) u ostvarenju alarma se čak koristi i *radno čekanje* (u petlji se čita vrijednost brojila dok ne poprimi željenu vrijednost, kao u primjeru `Chapter_01_Startup/02_Example_clock`). Isto se načelo može upotrijebiti i u drugim sustavima, kada se unutar jezgre ustanovi da nije isplativo vratiti se u dretvu koja će vrlo skoro biti prekinuta alarmom druge dretve (ili bi se alarm pojavio i prije dovršetka prebacivanja u dretvu).

Aktiviranje alarma u nekim sustavima može imati i dodatne funkcionalnosti, kao na primjer slanje određenog signala prema dretvi. Signali su posebni oblik asinkrone komunikacije između dretvi, ali i između jezgre i dretvi. Signali se intenzivno koriste na *UNIX* i izvedenim sustavima te POSIX usklađenim sustavima. Jedno ostvarenje signala prikazano je u odjeljku 11.5.

8.6. Nadzorni alarm

Ugrađeni sustavi trebaju biti građeni tako da mogu dugotrajno raditi bez vanjske intervencije. Zato se oni vrlo pažljivo projektiraju i izgrađuju. Međutim, uvijek postoji mogućnost da neka programska ili sklopovska komponenta zakaže ili da postoji greška u programu ili sklopu ili da se pojave neočekivani ulazi i situacije, kao primjerice kratkotrajni strujno/naponski poremećaj.

Najjednostavniji i najčešće primjenjivani *postupak oporavka od pogreške* za ugrađene sustave je *resetiranje cijelog sustava* (re-inicijalizacija i ponovno podizanje).

Problem je kako otkriti grešku te što onda napraviti.

Greške koje uzrokuju zaustavljanje sustava ili njegovih kritičnih dretvi može se otkriti i dodatnim sklopovljem – *nadzornim alarmom* (engl. *watchdog timer*). Za razliku od normalnog alarma koji šalje signale sustavu (ili nekoj dretvi), nadzorni alarm radi na suprotnom načelu: sustav "šalje signale" nadzornom alarmu.

Načelo rada sustava s nadzornim alarmom je u periodičkom poništavaju (*resetiranju*) alarma – alarmu se ponovno postavlja početni interval za odbrojavanje. Alarm odbrojava od neke zadane vrijednosti do nule. Ako on u tom periodu nije primio signal koji ga poništava, on će po isteku perioda resetirati cijeli sustav – aktivirati signal na RESET priključku procesora. Naime, sustav je tako građen da se u normalnom radu nadzorni alarm periodički poništava iz ključnih dijelova programa te nikada ne odbroji do kraja. U slučaju zastoja (kritične greške) alarm se

neće poništiti te kao jedan od mehanizama oporavka jest ponovno pokretanje cijelog sustava (*reset*).

Ostvarenje nadzornog alarma zahtijeva analizu i odabir periode u kojima nadzorni alarm treba signalizirati (poništiti) od strane programa koji se nadzire. Također, signalizaciju treba ugraditi u sam program, tj. treba odabrati dijelove kôda u koji treba ugraditi pozive za signalizaciju nadzornog alarma.

Primjer jednostavnog programa [Murphy, 2001] koji koristi nadzorni alarm, tj. periodički postavlja brojač nadzornog alarma na početnu vrijednost (svakim prolazom petlje) nalazi se u nastavku:

Primjer korištenja nadzornog alarma

```
uint16 volatile *pWatchdog = (uint16 volatile *) 0xFF0000;
int main ()
{
    hwinit();

    for (;;)
    {
        *pWatchdog = 10000;
        read_sensors();
        control_motor();
        display_status();
    }
}
```

Izvedbe nadzornog alarma mogu biti i složenije od prikazanog [Murphy, 2000]. Primjerice, ponekad je bitno detektirati i prerana poništavanja alarma (kada je poznat minimalni interval između dva poništavanja).

Za neke blaže greške, gdje zakaže samo pojedina dretva i to u prekidivom dijelu (kada su prekidi dozvoljeni), mogao bi se nadzorni alarm ostvariti korištenjem alarma prikazanih u odjeljku 8.4. Primjerice, dretva bi mogla početno postaviti alarm na 60 sekundi. U tijeku svog rada dretva bi svako malo poništila alarm, tj. odgodila vrijeme njegove aktivacije. Ako se dretva zaustavi na duže od 60 sekundi, alarm će se aktivirati i poduzeti određene akcije. Na primjer, preko prikladnog će sučelja signalizirati jezgri o pojavi kritične greške na koju ona mora reagirati prekidanjem i ponovnim pokretanjem dotične dretve (navedeno nije ostvareno u Benu).

8.7. Upravljanje objektima jezgre

Dinamička dodjela spremnika za potrebe objekata jezgre, tj. raznih opisnika može se napraviti sučeljem `kmalloc`. Međutim, s istim sučeljem traži se spremnik i za druge potrebe (npr. razne međuspremnike). U sustavu u kojem nema zaštite spremnika (a u prikazanom se ta zaštita uvodi tek na kraju), zbog grešaka u samim programima može zatajiti cijeli sustav. Primjerice, ako se zbog greške u programu prepiše opisnik nekog objekta jezgre može se srušiti cijeli sustav. Radi mogućnosti otkrivanja takvih grešaka u prikazanom sustavu koristi se nekoliko dodatnih struktura podataka i mehanizama opisanih u nastavku.

Svaki objekt jezgre koji se koristi iz programa dodatno se omata strukturom `kobject_t` koja sadrži kazaljku na objekt jezgre (pored ostalih pomoćnih elemenata) koji se postavlja u listu objekata `kobjects`. Također, svaki objekt jezgre (bez omotača) pored ostalih potrebnih podataka ima i jedinstveni identifikacijski broj u jezgri koji se dohvaća s `k_new_id` te po oslobađanju vraća s `k_free_id`. U svakom se pozivu jezgrine funkcije (u `DEBUG` načinu rada) provjerava je li kazaljka na objekt jezgre u toj listi.

Pogledajmo primjer navedenog mehanizma kod alarma. Stvaranje alarma započinje u funkciji `sys__timer_create`:

Isječak kôda 8.10. Chapter_04_Timer/03_Timers/kernel/time.c

```
500 int sys__timer_create(clockid_t clockid, sigevent_t *evp, timer_t *timerid)
501 {
502         ktimer_t *ktimer;
503         int retval;
504         kobject_t *kobj;
505 
506         SYS_ENTRY();
507 
508         ASSERT_ERRNO_AND_EXIT(
509         clockid == CLOCK_REALTIME || clockid == CLOCK_MONOTONIC, EINVAL);
510         ASSERT_ERRNO_AND_EXIT(evp && timerid, EINVAL);
511 
512         retval = ktimer_create(clockid, evp, &ktimer);
513         if (retval == EXIT_SUCCESS)
514         {
515                 kobj = kmalloc_kobject(0);
516                 kobj->kobject = ktimer;
517                 timerid->id = ktimer->id;
518                 timerid->ptr = kobj;
519         }
520 
521         SYS_EXIT(retval, retval);
522 }
```

U navedenoj se funkciji poziva `ktimer_create` koji stvara objekt jezgre koji se onda u ovoj funkciji omata s `kobject_t`. Opisnik koji se vraća u program sastoji se od kazaljke na objekt tip `kobject_t` te identifikacijski broj objekta. Obje se vrijednosti provjeravaju u idućim pozivima jezgrinih funkcija te u slučaju problema zaustavlja se daljnji rad (u idućim inkrementima u kojima se pojavljuje višedretvenost samo se takva se dretva zaustavlja, a ne i cijeli sustav kao ovdje). Provjera opisnika vidi se iz iduće funkcije `sys__timer_delete`:

Isječak kôda 8.11. Chapter_04_Timer/03_Timers/kernel/time.c

```
529 int sys__timer_delete(timer_t *timerid)
530 {
531         ktimer_t *ktimer;
532         int retval;
533         kobject_t *kobj;
534 
535         SYS_ENTRY();
536 
537         ASSERT_ERRNO_AND_EXIT(timerid, EINVAL);
538         kobj = timerid->ptr;
539         ASSERT_ERRNO_AND_EXIT(kobj, EINVAL);
540         ASSERT_ERRNO_AND_EXIT(list_find(&kobjects, &kobj->list),
541                                 EINVAL);
542 
543         ktimer = kobj->kobject;
544         ASSERT_ERRNO_AND_EXIT(ktimer && ktimer->id == timerid->id, EINVAL);
545 
546         retval = ktimer_delete(ktimer);
547 
548         kfree_kobject(kobj);
549 
550         SYS_EXIT(retval, retval);
551 }
```

Navedena zaštita nije idealna, ali možda dostatna za jednostavnije sustave.

Pitanja za vježbu 8

1. Koje zahtjeve programi postavljaju prema podsustavu upravljanja vremenom?
2. Što je to *alarm* u kontekstu upravljanja vremenom?
3. Koje je osnovno sklopovlje potrebno za upravljanje vremenom? Opišite rad takvog sklopa i kako se pomoću njega može ostvariti praćenje vremena i ostvarenje jednostavnog alarma.
4. Navedite POSIX sučelje za upravljanje vremenom. Koje operacije obavlja pojedino sučelje (u jezgri)?
5. Što je to *povratna vrijednost funkcije*, a što *oznaka greške* (engl. *error number*)?
6. Opišite moguće ostvarenje podsustava za upravljanje vremenom ako na raspolaganju stoji sučelje sloja arhitekture koje omogućava samo jedan alarm.
7. Što je to *nadzorni alarm*? Čemu služi i kako se koristi?

9. Korištenje naprava

9.1. Sučelje za korištenje naprava

Svaka je *ulazno-izlazna naprava* (skraćeno UI, engl. *input-output – IO*) posebna i traži poseban način rada i oblik komunikacije. Međutim, radi pojednostavljenja upravljanja napravama u operacijskim sustavima, uobičajeno je da se definira sučelje prema kojem se izgrađuju *upravljački programi* (engl. *device driver*). Izuzetak mogu biti naprave od posebnog značenja (primjerice sklop za prihvat prekida).

Prednosti definicije i korištenja sučelja su u:

- jednostavnoj zamjeni jedne naprave drugom u izvođenju sličnih operacija (primjerice izlaz/ispis može se prikazati na zaslonu ili poslati preko serijske veze ili korištenjem drugih protokola i veza je ostvariti na isti način, kroz isto sučelje, ali raznim napravama)
- lakšoj integraciji novih upravljačkih programa, jednostavnije ostvarenje podsustava jezgre za upravljanje napravama (svi upravljački programi se dodaju na isti način).

Sučelja prema napravama mogu se podijeliti na sučelja prema jezgri te sučelja prema programima. Sučelje prema jezgri ostvareno je upravljačkim programima (i sučeljem koje oni ostvaruju), dok je sučelje prema programima definirano jezgrom.

Načelo rada s napravama u ovom je sustavu zasnovano na izravnom korištenju upravljačkih programa. Funkcije upravljačkih programa pozivaju se pri zahtjevu za rad s napravom iz programa (preko jezgrenih funkcija), ali i pri zahtjevu za prekid koji izaziva naprava. Radi povećanja mogućnosti upravljanja sustavom, na svaki zahtjev za prekid prvo se poziva jezgrena funkcija koja provjerava povezanost prekida i naprave te potom koristi odgovarajuće funkcije upravljačkog programa za obradu prekida naprave. Budući da su naprave u pravilu značajno sporije od procesora uobičajeno je da se za rad s njima koriste međuspremnici. Međutim, ovdje se to nije napravilo u sloju jezgre već je to prepušteno upravljačkim programima.

9.1.1. Sučelje prema jezgri

Najjednostavnije sučelje za naprave (za izgradnju upravljačkih programa) uključuje funkciju za slanje podataka prema napravi te funkciju za čitanje podataka iz naprave.

U petom inkrementu[1], strukturom `device_t` (u datoteci `include/arch/device.h`) je definirano *sučelje* upravljačkih programa sa sljedećim elementima (podacima i kazaljkama na funkcije):

- `dev_name` – ime naprave (koristi se pri dohvatu opisnika naprave)
- `init` – funkcija za inicijalizaciju naprave
- `destroy` – funkcija za uklanjanje naprave (ona se programski onemogućava, tj. isključuje)
- `send` – funkcija za slanje podataka prema napravi
- `recv` – funkcija za čitanje podataka s naprave
- `irq_num` – broj prekida koji izaziva naprava (ako izaziva)
- `irq_handler` – funkcija za obradu prekida koji je izazvala naprava
- `callback` – jezgrena funkcija koju treba pozvati (ako je definirana) iz obrade prekida

[1] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_05_Devices`.

naprave (iz `irq_handler` poziva se `callback`)[2]

- `flags` – razne zastavice koje mogu pobliže definirati posebna ponašanja naprave (načini rada i slično)
- `params` – kazaljka koju upravljački program naprave može koristiti za svoje interne potrebe.

Za svaku napravu koju se želi dodati u sustav treba ostvariti upravljački program prema prethodnom sučelju. Ostvareni upravljački programi u Benu smješteni su u direktorij `arch/i386/drivers` i uključuju podršku za ispis na zaslon (`vga_text`), korištenje tipkovnice (`i8042`) te komunikaciju preko serijske veze (`uart`). U istom se direktoriju nalaze i upravljački programi za sklop za prihvat prekida (`i8259`) te brojilo (`i8253`), ali se oni ne koriste preko `device_t` sučelja već preko `arch_ic_t` te `arch_timer_t` sučelja, s obzirom na to da su to sklopovi s posebnim namjenama.

Naprave, tj. njihovi upravljački programi se mogu uključiti ili ne uključiti u izlazni sustav. Mehanizam za upravljanje uključivanjem ili isključivanjem pojedine naprave u postupku izgradnje jesu makroi, tj. varijable definirane (ili ne) u `config.ini` datoteci. Primjerice, makroom `UART` se uključuje upravljački program za *uart* sklop jer u datotekama `uart.h` i `uart.c` postoje provjere:

```
#ifdef UART
...
#endif
```

koje će opcionalno uključiti sadržaje tih datoteka u prevođenju.

Kao i za ostala sučelja do sada, u datotekama s izvornim kôdom upravljačkih programa definiraju se globalne varijable sa sučeljem (`vga_text_dev` u `vga_text.c`, `i8042_dev` u `i8042.c` te `uart_com1` u `uart.c`).

Dodavanje nove naprave u ovaj sustav, tj. njenog upravljačkog programa zahtijeva:

1. izradu upravljačkog programa prema sučelju `device_t`, uključujući definiranje globalne varijable koja to sučelje sadrži
2. uključivanje tog upravljačkog programa u postupak izgradnje definiranjem odgovarajućeg makroa u `config.ini` te dodavanjem vrijednosti u varijablu `DEVICES` i njegovo korištenje u izvornim kôdovima upravljačkog programa
3. dodavanjem imena globalne varijable (ostvarene pod 1.) u popis naveden u `config.ini`, tj. varijabli `DEVICES_DEV`
4. definiranje imena kojim će se ta naprava identificirati u sustavu.

Ako bi bilo potrebno postaviti posebne parametre pri inicijalizaciji, onda se zadnja dva koraka preskaču a naprava inicijalizira izravno u kodu (preskače se automatska inicijalizacija i dodavanje naprave u popis naprava).

U datoteci `config.ini` je dodan odjeljak posvećen upravljanju napravama prema spomenutom načelu.

Isječak kôda 9.1. Chapter_05_Devices/03_Serial_port/arch/i386/config.ini

```
24 # Devices
25 #-------------------------------------------------------------------------------
26 #"defines" (which device drivers to compile)
27 DEVICES = VGA_TEXT I8042 I8259 I8253 UART
```

[2] S obzirom na to da jezgra definira koju njenu funkciju treba pozvati u obradi prekida, ovaj se oblik poziva naziva i *povratni poziv* (engl. *callback*). Slični se mehanizmi mogu primijetiti i kod korištenja sučelja prema napravama (sučelje definira adrese funkcija koje treba pozivati za određene operacije), kod obrade prekida, korištenje alarma i signala.

```
28
29 #devices interface (variables implementing device_t interface)
30 DEVICES_DEV = dev_null vga_text_dev uart_com1 i8042_dev
31
32 #interrupt controller device
33 IC_DEV = i8259
34
35 #timer device
36 TIMER = i8253
37
38 #initial standard output device (while "booting up")
39 K_INITIAL_STDOUT = uart_com1
40 #K_INITIAL_STDOUT = vga_text_dev
41
42 #standard output for kernel function (for kprint) - device name
43 K_STDOUT = COM1
44 #K_STDOUT = VGA_TXT
45
46 #standard output and input devices for programs
47 U_STDIN = COM1
48 U_STDOUT = COM1
49 U_STDERR = COM1
50 #U_STDIN = i8042
51 #U_STDOUT = VGA_TXT
52 #U_STDERR = VGA_TXT
```

Osim definiranja koje naprave uključiti u sustav, u gornjim linijama su definirane i naprave koje će se koristiti kao *standardni izlaz* i *standardni ulaz*, s obzirom na to da postoje dvije naprave za svaki (tipkovnica i serijski pristup za ulaz, zaslon i serijski pristup za izlaz).

Korištenjem mogućnosti alata *Make* u datoteci `Makefile` se izgrađuju varijable `DEV_VARS` i `DEV_PTRS` koje se koriste pri inicijalizaciji naprava u `kernel/device.c`.

Isječak kôda 9.2. Chapter_05_Devices/03_Serial_port/kernel/device.c

```
27 /*! Initialize 'device' subsystem */
28 int k_devices_init()
29 {
30         extern device_t DEVICES_DEV; /* defined in arch/devices, Makefile */
31         device_t *dev[] = {DEVICES_DEV_PTRS, NULL};
32         kdevice_t *kdev;
33         int iter;
34
35         list_init(&devices);
36
37         for (iter = 0; dev[iter] != NULL; iter++)
38         {
39                 kdev = k_device_add(dev[iter]);
40                 k_device_init(kdev, 0, NULL, NULL);
41         }
42
43         return 0;
44 }
```

Radi mogućnosti ostvarenja kontrole, iz jezgre i iz programa naprave se ne koriste izravno preko njihovih sučelja, već su napravljene dodatne jezgrene funkcije preko kojih se naprave trebaju koristiti, a koje omogućuju ugradnju potrebne kontrole[3]. Za te potrebe definirana je nova struktura u jezgri `kdevice_t` (datoteka `kernel/device.h`), koja sadrži elemente i varijable:

- `dev` – sučelje upravljačkog programa (`device_t` ostvarenog u sloju *arch*)
- `id` – jedinstven identifikator objekta jezgre

[3]Načelo slojevite izgradnje operacija sustava je prethodno objašnjeno kod ostvarenja alarma, u odjeljku 8.4.1.

- `flags` – označava je li naprava "otvorena" u nedjeljivom načinu, tj. može li se opet otvoriti od strane nekog drugog objekta
- `ref_cnt` – brojač procesa koji koriste napravu
- `descriptors` – lista opisnika koji sadrže kazaljku na ovu napravu
- `list` – koristi se za ostvarenje liste upravljačkih programa (svi se svrstavaju u jednu zajedničku listu pri inicijalizaciji sustava).

Inicijalizacija svih naprava funkcijom `k_devices_init` obavlja se pri pokretanju sustava iz datoteke `kernel/startup.c`.

Isječak kôda 9.3. Chapter_05_Devices/03_Serial_port/kernel/startup.c

```
46          k_devices_init();
47
48          /* switch to default 'stdout' for kernel */
49          k_stdout = k_device_open(K_STDOUT, O_WRONLY);
```

Korištenje naprave mora započeti spajanjem na napravu – "ostvaranjem naprave". Pri spajanju na napravu potrebno je navesti jedinstveno ime naprave. U izvornom kôdu, umjesto izravnog navođenja imena naprava (koja su definirana u upravljačkim programima) koriste se varijable definirane u datoteci `config.ini` radi jednostavnijeg i centraliziranog podešavanja sustava (odabir koje će se naprave koristiti za standardni ulaz i izlaz). Naprava može biti dijeljena, kada ju se može otvoriti i više puta (primjerice od strane različitih dretvi u idućim koracima) ili nedjeljiva, kada ju se može otvoriti samo jednom istovremeno.

9.1.2. Sučelje prema programima

Jezgra koristi sučelje upravljačkih programa `device_t`, ali se to sučelje ne koristi izravno već preko funkcija `k_device_*` radi dodatne provjere parametara i mogućnosti naprave. Primjerice, ako je naprava izlazna, sučelje `.send` ne mora biti ostvareno ni zadano u sučelju (može biti `NULL`) te bi pokušaj poziva uzrokovao grešku sustava.

To se sučelje koristi i pri zahtjevima iz programa, ali uz dodatne provjere i prilagodbe u dodatnim funkcijama `sys__*`.

Odabrano sučelje za rad s napravama iz programa je jednako sučelju za rad s datotekama jer se slične operacije obavljaju u oba slučaja: *otvaranje*, *čitanje*, *pisanje* te *zatvaranje*. Razlika je u mogućnosti pomicanja kazaljke trenutnog položaja u datoteci koji nema smisla za naprave.

Operacije za korištenje naprava iz programa (dodatno prilagođene u `api/stdio.c`) su "uobičajene" funkcije: `open`, `close`, `read` i `write`. Dodatno, pomoću prethodnih funkcija su ostvarene i uobičajene funkcije za ispis na standardni izlaz (zaslon ili serijsku vezu) `printf` te za čitanje sa standardnog ulaza (tipkovnice ili serijske veze) `getchar`.

9.2. Zaslon kao naprava

Prvo korištenje sučelja `device_t` prikazano je nad upravljačkim programom za ispis na *zaslon*. Postojeći upravljački program, koji je koristio `console_t` sučelje, prepravljen je tako da koristi `device_t` sučelje. S obzirom na to da je ispis na zaslon jednosmjerna operacija, zapravo se koristi samo funkcija `send`. Podaci koji se šalju su naravno znakovi za ispis (u ASCII obliku), ali i naredbe (za brisanje cijelog zaslona te za pomicanje značke).

Da bi se dobio uvid u moguća ponašanja naprava osim jednostavne izlazne naprave za ispis na zaslon, u nastavku su detaljnije opisane dvije naprave: tipkovnica i serijski pristup.

9.3. Tipkovnica

Tipkovnica je jedna od ulaznih naprava računalnog sustava. Osim vanjskog dijela koji korisnici izravno koriste – tipke, tipkovnica ima upravljački sklop koji privremeno pohranjuje stisnute tipke te generira zahtjeve za prekide na svaki događaj pritiska i otpuštanja tipki.

Stanja tipki se periodički očitavaju (*skeniraju*) pod utjecajem internog upravljačkog sklopa tipkovnice (*kodera*). Ako se ustanovi da je neka tipka promijenila stanje (pritisnuta, otpuštena ili zadržana), podatkovni se paket šalje upravljačkom sklopu tipkovnice koji se obično nalazi na matičnoj ploči računala. Podatkovni paket koji se šalje naziva se *očitani kôd* (engl. *scan code*). Upravljački sklop na tipkovnici (koder) saznaje koji očitani kôd odgovara određenoj tipki pomoću mape znakova koja se nalazi u njegovom ROM-u. Kada upravljački sklop tipkovnice (na matičnoj ploči) primi očitani kôd, sprema ga u ulazni spremnik i signalizira prekid preko ulaza IRQ1 (2. ulaz sklopa *Intel 8259*). U obradi prekida treba pročitati što se novo dogodilo i proslijediti događaj odgovarajućem programu (koji očekuje ulaz preko tipkovnice).

Postoje dvije vrste očitanih kôdova: kôd pritiskanja (engl. *make code*) i kôd otpuštanja (engl. *break code*). Kôd pritiskanja se šalje kada je tipka pritisnuta ili zadržana dolje, a kôd otpuštanja se šalje kada je tipka otpuštena. Svaka tipka ima svoj jedinstveni kôd pritiskanja i kôd otpuštanja. Ako se neka tipka pritisne i drži stisnutom, tada se kôd pritiskanja te tipke šalje računalu sve dok se tipka ne otpusti ili se neka druga tipka ne stisne.

Postoje različite grupe očitanih kôdova za različite tipkovnice:

- grupa 1 (XT očitani kôdovi)
- grupa 2 (AT očitani kôdovi)
- grupa 3 (PS/2 očitani kôdovi).

Očitane kôdove grupe 1 koristila su starija računala (slika 9.1.). Moderne tipkovnice obično koriste grupu 2. Kako bi se sačuvala usklađenost s ranijim programima upravljač tipkovnice može pretvarati kôdove grupe 2 u kôdove grupe 1.

Slika 9.1. XT kôdovi pritiskanja

Upravljač tipkovnice – sklop *Intel 8042* sadrži sljedeće registre: ulazni spremnik, izlazni spremnik, statusni registar i upravljački registar. Tablica 9.1. prikazuje U/I adrese upravljačkog sklopa tipkovnice te koje se operacije nad njima mogu obavljati.

Očitani kôd može se pročitati s U/I adrese `0x60`. Statusni oktet čita se s adrese `0x64`. Naredba se šalje pisanjem naredbenog okteta na adresu `0x64`. Popis naredbi može se pronaći u literaturi [Tipkovnica].

U pojednostavljenom prikazu (koji se u nastavku koristi), tipkovnica se može prikazati sklopom

Tablica 9.1. Adrese i operacije upravljačkog sklopa tipkovnice

adresa	operacije	opis
`0x60`	čitanje	čitanje ulaznog spremnika
`0x60`	pisanje	pisanje u ulazni spremnik
`0x64`	čitanje	čitanje statusnog registra
`0x64`	pisanje	slanje naredbi

koji pohranjuje nekoliko zadnjih akcija nad tipkovnicom (stisaka i otpuštanja tipaka), kao što to prikazuje slika 9.2.

Slika 9.2. Pojednostavljeni prikaz sustava s tipkovnicom

Za svaku akciju nad tipkovnicom sklop generira zahtjev za prekid (broj 33 s obzirom na to da je spojen na ulaz IR1 sklopa *Intel 8259*). Čitanjem registra upravljačkog sklopa tipkovnice dobiva se prvi nepročitani očitani kôd. Prije toga potrebno je provjeriti ima li novih nepročitanih događaja s tipkovnicom (preko drugog, *statusnog registra*). Za sve navedeno osim obrade prekida, sučelje je vrlo jednostavno – treba pročitati sadržaj registara (instrukcijom `in`). U obradi prekida te događaje treba barem donekle interpretirati.

Iako je preko sučelja `device_t` moguće ostvariti i povratni poziv prema jezgri i time ostvariti i mogućnost zaustavljanja dretve pri operaciji čitanja (zaustaviti dretvu dok se nešto ne pojavi u međuspremniku, tj. dok se nešto ne stisne na tipkovnici) to u ovom ostvarenju nije napravljeno. Ista se funkcionalnost može napraviti i u jezgri, uvidom u povratnu vrijednost funkcija `send` i `recv`.

Upravljački program tipkovnice obavlja dvije osnovne operacije:

- u obradi prekida čita kôdove pritisaka i otpuštanja tipki iz sklopa *Intel 8042*, obavlja osnovnu obradu (pretvorbu u kod ASCII) te ih stavlja u svoj (programski) međuspremnik (koji se nalazi u radnom spremniku) te
- u zahtjevima za čitanjem, koji dolaze od programa (ili jezgre), vraća prvi nepročitani znak iz svog međuspremnika (prema slici 9.2.).

Upravljački program tipkovnice ostvaren u Benu u obradi prekida dohvaća zadnju akciju te se u ovisnosti o njoj poduzimaju neke aktivnosti:

- ako je to bio pritisak na običnu tipku, njen se ASCII kôd stavlja u zaseban međuspremnik (u radnom spremniku, ne sklopu tipkovnice);
- ako je to bio događaj otpuštanja obične tipke – događaj se ignorira;
- ako je stisnuta ili otpuštena neka od posebnih funkcijskih tipki (*Ctrl*, *Shift*, *Alt*, ...), onda

se kôd te tipke (interno dodijeljen u `arch/i386/drivers/i8042.h`) stavlja u međuspremnik te se ažurira posebna varijabla koja označava stanje tih tipki (stisnuto ili ne).

Interpretacija kombinacije tipki ponekad predstavlja zasebno značenje. Na primjer, ako se tipka *Shift* drži stisnutom dok se pritišće tipka *H*, zadnji se događaj interpretira kao unos velikog slova *H*, a ne malog *h*. Doseg ostvarenog upravljanja tipkovnicom jest samo u razlikovanju unosa u ovisnosti o dodatnom držanju tipki lijeve i desne tipke *Shift* ili aktivacija *CapsLock* tipke. Ostale se tipke (obične i funkcijske) samo upisuju u međuspremnik. Ako je neka druga kombinacija tipki bitna, to će morati zaključiti program koji će zatražiti dohvat unesenih događaja (pritisaka tipki).

Ako je tako zadano, svaki pritisak na obične tipke pokazuje se i ispisom odgovarajućeg znaka na zaslon (zastavica `ECHO_ON`).

Slojevitost u korištenju naprave vidljiva je i na primjeru tipkovnice, krećući od programa, sloja *api*, sloja jezgre do upravljačkog programa. Isječci kôda koji to prikazuju nalaze se u nastavku.

Isječak kôda 9.4. Chapter_05_Devices/03_Serial_port/programs/keyboard/keyboard.c

```
15          do {
16                  if ((key = getchar()))
17                          printf("Got: %c(%d)\n", key, key);
18                  else
19                          nanosleep(&t, NULL);
20          }
21          while (key != '.');
```

Isječak kôda 9.5. Chapter_05_Devices/03_Serial_port/api/stdio.c

```
111 int getchar()
112 {
113         int c = 0;
114
115         read(_stdin, &c, sizeof(int));
116
117         return c;
118 }
```

Isječak kôda 9.6. Chapter_05_Devices/03_Serial_port/api/stdio.c

```
85  ssize_t read(int fd, void *buffer, size_t count)
86  {
87          if (    fd < 0 || fd >= MAX_USER_DESCRIPTORS ||
88                  !std_desc[fd].id || !std_desc[fd].ptr || !buffer || !count)
89          {
90                  set_errno(EBADF);
91                  return EXIT_FAILURE;
92          }
93
94          return sys__read(&std_desc[fd], buffer, count);
95  }
```

Isječak kôda 9.7. Chapter_05_Devices/03_Serial_port/kernel/device.c

```
301 int sys__read(descriptor_t *desc, void *buffer, size_t size)
302 {
303         return read_write(desc, buffer, size, TRUE);
304 }
305 int sys__write(descriptor_t *desc, void *buffer, size_t size)
306 {
307         return read_write(desc, buffer, size, FALSE);
308 }
309
310 static int read_write(descriptor_t *desc, void *buffer, size_t size, int op)
311 {
```

```
312         kdevice_t *kdev;
313         kobject_t *kobj;
314         int retval;
315
316         SYS_ENTRY();
317
318         ASSERT_ERRNO_AND_EXIT(desc && buffer && size > 0, EINVAL);
319
320         kobj = desc->ptr;
321         ASSERT_ERRNO_AND_EXIT(kobj, EINVAL);
322         ASSERT_ERRNO_AND_EXIT(list_find(&kobjects, &kobj->list),
323                                 EINVAL);
324         kdev = kobj->kobject;
325         ASSERT_ERRNO_AND_EXIT(kdev && kdev->id == desc->id, EINVAL);
326
327         /* TODO check permission for requested operation from opening flags */
328
329         if (op)
330                 retval = k_device_recv(buffer, size, kobj->flags, kdev);
331         else
332                 retval = k_device_send(buffer, size, kobj->flags, kdev);
333
334         if (retval >= 0)
335                 SYS_EXIT(EXIT_SUCCESS, retval);
336         else
337                 SYS_EXIT(-retval, EXIT_FAILURE);
338 }
```

Isječak kôda 9.8. Chapter_05_Devices/03_Serial_port/kernel/device.c

```
143 int k_device_recv(void *data, size_t size, int flags, kdevice_t *kdev)
144 {
145         int retval;
146
147         if (kdev->dev.recv)
148                 retval = kdev->dev.recv(data, size, flags, &kdev->dev);
149         else
150                 retval = EXIT_FAILURE;
151
152         return retval;
153 }
```

9.4. Serijska veza

Načelno, upravljanje i korištenje serijske veze vrlo je slično kao i kod tipkovnice. Razlika je u sklopovlju, više statusnih i upravljačkih registara te dvosmjerna komunikacija. Komunikacija serijskom vezom jest serijska, tj. bitovi se preko jedne žice prenose slijedno, jedan po jedan, ali grupirani u obliku jedne poruke. Poruka osim korisne informacije sadržava početni "start bit", bit za provjeru pariteta te završni "stop bit" (ili dva). Tipičan okvir jedne poruke može se prikazati slikom 9.3.

Slika 9.3. Poruka 01000101_2 duljine 8 bita, parnog pariteta uz 2 stop bita

Sklop koji se koristi jest *8250 UART* (ili noviji kompatibilni) te su upravljački programi za njega pripremljeni. Sklop ima nekoliko ulaznih i nekoliko izlaznih registara na adresama koje

započinju adresom sklopa (primjerice `0x3F8` za `COM1`).

U datoteci `arch/i386/drivers/uart.h` nalazi se popis uobičajenih adresa određenih izlaza serijskih veza te odmaka za pojedine registre sklopa.

Isječak kôda 9.9. Chapter_05_Devices/03_Serial_port/arch/i386/drivers/uart.h

```
39 #define COM1_BASE       0x3f8
40 #define COM2_BASE       0x2f8
41 #define COM3_BASE       0x3e8
42 #define COM4_BASE       0x2e8
43
44 #define THR     0       /* Transmitter Holding Buffer */
45 #define RBR     0       /* Receiver Buffer */
46 #define DLL     0       /* Divisor Latch Low Byte */
47 #define IER     1       /* Interrupt Enable Register */
48 #define DLM     1       /* Divisor Latch High Byte */
49 #define IIR     2       /* Interrupt Identification Register */
50 #define FCR     2       /* FIFO Control Register */
51 #define LCR     3       /* Line Control Register */
52 #define MCR     4       /* Modem Control Register */
53 #define LSR     5       /* Line Status Register */
54 #define MSR     6       /* Modem Status Register */
55 #define SR      7       /* Scratch Register */
```

Neki registri dijele istu adresu ako su samo ulazni ili izlazni ili ako se aktiviraju posebnim zapisima u upravljačkim registrima. Osnovni registri za slanje (`THR`) i primanje (`RBR`) podataka nalaze se na početnoj adresi (s odmakom jednakim nuli). Osim njih, od bitnijih registara treba izdvojiti registre koji određuju brzinu prijenosa (`DLL` i `DLM`), registre koji upravljaju međuspremnicima sklopa (`FCR`) u registre za određivanje načina dojave posebnih stanja (`IER` i `IIR`).

Uobičajene postavke serijske veze označene s `8N1` definiraju duljinu riječi od 8 bita, odsustvo bita pariteta (bez pariteta) te jedan "stop bit". Dodatni parametar definira brzinu prijenosa u *baudima* (znakovima u sekundi). Uobičajene vrijednosti su od 9600 do 115200.

S obzirom na veći broj registara za upravljanje, prikaz kôda bi trebao uključivati znatne opise. Zato je opis ovdje izostavljen. Specifičnost sučelja za korištenje serijske veze jest što se u varijabli `.params` pohranjuje kazaljka na strukturu koja opisuje postavke veze te sadrži ulazni i izlazni međuspremnik.

Pitanja za vježbu 9

1. Što je to upravljački program (naprave)?
2. Koje osnovne elemente mora imati sučelje upravljačkog programa?
3. Kako jezgra treba upravljati napravama?
4. Što je to očitani kôd (engl. *scan code*)?
5. Kako su idejno ostvareni upravljački programi za tipkovnicu i serijsku vezu?
6. Što su to funkcije s *povratnim pozivom* (engl. *callback*)? Gdje se one koriste u prikazanom sustavu?

10. Naredbena ljuska

Interaktivnost sustava može se iskazati programom koji preko ulaznih naprava (primjerice tipkovnica, miš) čita naredbe koje zadaje korisnik te prema njima djeluje. Međutim, jedan program najčešće pruža tek nekoliko povezanih operacija koje obavljaju određenu aktivnost za korisnika. Druge operacije i aktivnosti ostvarene su u drugim programima, koje tada treba moći pokrenuti. Pokretanje programa od sustava zahtijeva zauzimanje spremničkog prostora za program (instrukcije, podatke, gomilu, stog), učitavanje programa u spremnik, stvaranje opisnika procesa i dretve te uvrštavanje tih opisnika u odgovarajuće liste.

Radi jednostavnijeg korištenja sustava s više programa, može se napraviti dodatni (pomoćni) program koji preko jednostavnog sučelja olakšava korisniku pokretanje osnovnih programa sustava. Primjerice ako sustav posjeduje pet osnovnih programa koje obični korisnik pokreće, može se napraviti dodatni program koji se automatski pokreće sa sustavom, a koji će u grafičkom sučelju prikazati pet gumba za pokretanje svakog od tih programa. Problem ovakvog rješenja je u neprilagodljivosti. Svakim dodavanjem novih programa namijenjenih i korisniku, treba mijenjati i početni program sa svojim sučeljem.

Općenitije rješenje, koje se koristi u operacijskim sustavima opće namjene, omogućuje pokretanje svih programa, nekih preko sučelja, prečaca, izbornika, a drugih izravno zadavanjem puta do datoteke sa željenim programom. Manje zahtjevnim korisnicima ovakav način pokretanja može biti suviše složen te si oni najčešće korištene programe dodaju u brzo dostupne izbornike i mape s prečacima.

Najprimitivniji način pokretanja programa, prikazan u ovom poglavlju, jest pokretanje programa *zadavanjem naredbi u naredbenoj ljusci* (engl. *command prompt* ili *command shell* ili samo *shell*), tj. utipkavanjem imena programa. Iako primitivan po načinu pokretanja sa strane korisnika, potrebne operacije sustava za pokretanje programa su gotovo identične drugim načinima pokretanja (iz grafičkog sučelja).

Ljuska je zapravo "običan" program koji pokreće druge programe. Zato se njen rad može opisati cikličkim poslom:

```
ponavljaj
{
    ulaz = pročitaj naredbu s ulaza (primjerice tipkovnice);
    {naredba, parametri} = obradi_ulaz (ulaz);
    ako ( "naredba" je valjana/postojeća naredba )
        pokreni (naredba, parametri);
    inače
        dojavi grešku ("Nepostojeća naredba/program!");
}
dok ( naredba != "kraj" );
```

Zadana naredba može biti *ugrađena naredba ljuske* ili ime drugog programa koji treba pokrenuti. Primjerice, naredba "pomoć" bi mogla ispisati sve postojeće naredbe ljuske s uputama o korištenju. Ako je naredba ime drugog programa, ljuska ga mora znati pronaći i pokrenuti. Ako sustav ima neki oblik datotečnog sustava, onda bi ljuska trebala potražiti postoji li zadani program i ako postoji pokrenuti ga (sučeljem operacijskog sustava). Ako tako nešto ne postoji, ljuska bi trebala znati koji sve programi postoje i na koje se načine pokreću.

U okviru kôda za ovo poglavlje[1] prikazano je jedno ostvarenje ljuske koja može pokrenuti (na zahtjev) jedan od ugrađenih programa. S obzirom na to da datotečni sustav ne postoji, u početku (do `Chapter_08_Processes`) "programi" su zapravo funkcije koje ljuska neizravno

[1] Izvorni kôdovi koji se koriste u ovom poglavlju, odnosno datoteke čiji se sadržaj koristi, nalaze se u direktoriju `Chapter_06_Shell`.

poziva (korištenjem kazaljki). Ipak, već sada je pripremljena podloga za modularizaciju i opcionalno uključivanje programa (tj. funkcija) u sustav. Programi koje treba uključiti u sustav moraju se "ručno" pobrojati u zasebnom polju ili izravno u kôdu (za faze 01-03 u datoteci `include/api/prog_info.h`) ili preko datoteke s postavkama (u datoteci `config.ini` za fazu 04).

Ljuska je ostvarena kao dodatni program u `programs/shell/shell.c`.

Popis svih postojećih programa nalazi se strukturi `cmd_t` koja se u početku ručno ispunjava (`include/api/prog_info.h`) a kasnije poluautomatski stvara korištenjem odabranih programa (`Makefile`).

Isječak kôda 10.1. Chapter_06_Shell/04_Makepp/programs/shell/shell.c

```
13 typedef struct _cmd_t_
14 {
15         int (*func)(char *argv[]);
16         char *name;
17         char *descr;
18 }
19 cmd_t;
```

U početnoj fazi (`01_Shell`) ljuska raspoznaje obične naredbe (prva riječ do razmaka ili drugih graničnika), uspoređuje ih s postojećim imenima programa te ako nađe odgovarajući pozove zadanu funkciju (ostvarenu ili u samoj ljusci ili u drugim programima direktorija `programs`).

Iduća faza (`02_Arguments`) donosi podršku za obradu zadane naredbe i njeno raščlanjivanje na naredbu i argumente. Argumenti se u funkcije šalju kao polje kazaljki na niz znakova (svaki argument je jedan niz znakova). Zadnji element polja mora biti `NULL`. Načelo predaje argumenata je slično (ali nije isto) kao i kod funkcije `main` običnog C programa (`int main (int argc, char *argv[])`). Razlika je što nema zasebnog argumenta koji kaže koliko ima argumenata (`argc`) već se to mora doznati pretraživanjem polja i pronalaskom polja s vrijednošću `NULL` (koja može biti i na početku ako nema argumenata).

Treća faza (`03_Programs`) je usmjerena na dodatno odvajanje slojeva jezgre i programa dodavanjem strukture `prog_info_t` koja sadrži informacije o programima, odnosno iz jezgre se više ne poziva izravno početna funkcija već se poziva funkcija `prog_init` (ostvarena u `api/prog_info.c`) koja (dodatno) inicijalizira okolinu za programe (standardni ulaz i izlaz, inicijalizacija gomile) te pokreće početni program definiran varijablom `PROG_START_FUNC` u datoteci `config.ini`. Početni program više nije definiran u kôdu (`kernel/startup.c`) već u datotekama s postavkama.

U četvrtoj fazi (`04_Makepp` – *make++*), pokazan je način zasebnog prevođenja pojedinih dijelova sustava, tj. odvojeno je prevođenje jezgre i zasebno svakog programa. Razlog takve potrebe može biti u zasebnim (drukčijim) parametrima pri pokretanju (druge zastavice). Postavke se i dalje nalaze u datoteci `config.ini` dok je početni `Makefile` interno podijeljen u tri dijela: zajednički dio, dio za prevođenje jezgre te dio za prevođenje programa. Iako se takav pristup koristi u nastavku (ostalim poglavljima), on je možda najsloženiji dio datoteke `Makefile` te se njegovo razumijevanje i korištenje (promjena) ne traži od studenata, već je samo dan kao primjer naprednijih mogućnosti.

Posebnost prevođenja programa jest u korištenju funkcija koje se pozivaju u početnoj fazi analize `Makefile` datoteke. Naime, umjesto izravnog navođenja kako se prevodi svaki od programa, definirana je struktura podataka (varijable) te korištenjem funkcije obavljeno generiranje kôda koji se onda, u drugom koraku koristi pri prevođenju. Načelo je prikazno idućim primjerom.

Makefile1

```
VARS = prvi drugi treci

sve: $(VARS)
        @echo Sve gotovo

define FUNC
$(1):
        @echo Radim $(1)
endef

$(foreach var,$(VARS),$(eval $(call FUNC,$(var))))
```

Navedeni primjer je po funkcionalnosti jednak s idućim (`Makefile2`), bez funkcije, u koji se zapravo `Makefile1` prevodi u prvom prolazu.

Makefile2

```
VARS = prvi drugi treci
sve: $(VARS)
        @echo Sve gotovo
prvi:
        @echo Radim prvi
drugi:
        @echo Radim drugi
treci:
        @echo Radim treci
```

Prevođenje programa preko funkcije u datoteci `Makefile` je utoliko složenije što se unutar `define/endef` nalazi više toga i sam parametar se koristi i kao ime varijable, a ne samo kao vrijednost. Primjerice, kada se šalje `hello`, on se osim kao ime koristi i kao ime varijable koja ima dva elementa `{hello_world, programs/hello_world}`. Tim elementima se u funkciji `PROGRAM_TEMPLATE` pristupa s `$(word 1,$($(1)))` i `$(word 2,$($(1)))` (prvi i drugi element varijable `$($(1))`).

U većini projekata je ipak uobičajeno da se za odvojene komponente (module, podsustave, programe) koriste odvojene datoteke `Makefile`, po jedna za komponentu (direktorij u kojem se ona nalazi). Osnovni razlog tome je u znatno većim mogućnostima prilagodbe prevođenja za tu komponentu.

10.1. Ostvarenje ugradbenih sustava do sada prikazanim postupcima

Mnogi jednostavniji sustavi se mogu ostvariti do sada prikazanim postupcima, operacijama, algoritmima. Natuknice o nekim mogućnostima/potrebama za razvoj takvih sustava prikazane su u nastavku.

- razvojni alati
  - besplatni
  - komercijalni
- programiranje na ‘niskoj’ razini
  - asembler (posebne naredbe procesora)
  - upravljački programi (upravljanje napravama)
  - prekidi

- načini ostvarenja upravljanja
  - upravljačka petlja
  - korištenje 'operacijska sustava'
    - prekidni podsustav
    - vrijeme i alarmi
    - dinamičko upravljanje spremnikom (gomila/heap)
    - korištenje naprava preko sučelja (preko OS-a)
- operacijski sustav
  - koristiti neki postojeći:
    - besplatni (prednost = besplatan)
    - komercijalni (prednost = podrška)
  - izgrađen za potrebe 'zadanog' sustava (projekta na kojem se radi)
    - sastoji se samo od potrebnih komponenata
    - potpuniji (sadrži i komponente koje nisu potrebne za trenutni projekt, ali mogu biti korisne za nadogradnju ili druge projekte)
- složeniji ugrađeni sustavi
  - operacijski sustav koji se sastoji od do sada prikazanih komponenata ...
    - prekidni podsustav, upravljački programi, vrijeme, alarmi, dinamičko upravljanje spremnikom, nadziran pristup napravama, ljuska
  - ...je dostatan za mnoge sustave
  - ipak, za složenije sustave koji upravljaju s više naprava/procesa, ostvarenje upravljanja može biti složeno ili i nemoguće za ostvariti
  - višedretvenost TADA može pojednostaviti/omogućiti ostvarenje

Pitanja za vježbu 10

1. Čemu služi naredbena ljuska?
2. Koji računalni sustavi trebaju neki oblik korisničkog sučelja, a koji ne?
3. Kako se zadaju naredbe u naredbenoj ljusci?
4. Kako se predaju parametri programima?
5. Kako se iz programa koriste parametri (primjerice u običnoj `main` funkciji C programa)?

11. Višedretvenost

11.1. Uvodna razmatranja

Računalni sustavi koriste se za upravljanje nekim procesima (iz okoline) ili izvođenjem nekih korisniku potrebnih zadaća (programa). Upravljanje s više aktivnosti (zadaća, procesa) može se programski ostvariti na nekoliko načina. Ako se upravljanje može svesti na obradu događaja koje izazivaju vanjski procesi, tada se sva upravljačka logika može raspodijeliti u procedure koje obrađuju te događaje – u prekidne potprograme. Ako upravljanje traži periodičko očitanje stanja procesa te reakciju na očitanja, upravljanje se može ugraditi u obradu prekida sata ili izvesti programski, na način da se očitavaju svi upravljani procesi iz istoga kôda (periodičko "prozivanje").

Navedeni načini upravljanja, koji se mogu ostvariti s do sada prikazanim podsustavima, pogodni su samo za jednostavnije sustave. U složenijim bi sustavima navedeni postupci postali suviše složeni, teško ostvarivi i vrlo teški za održavanje, otkrivanje grešaka, nadograđivanje i slično. Logika upravljanja koja se mora ugraditi u druge podsustave ili "zajedničke" upravljačke programe postaje suviše složena i glavni je problem ostvarenja takvih načina upravljanja.

Značajno jednostavnije upravljanje postiže se odvajanjem upravljačkog programa u nezavisne entitete – *zadatke* (engl. *task*), od kojih se svaki brine za jedan vanjski proces. Za upravljanje pojedinim vanjskim procesima upravljačka je logika sva na jednom mjestu, razumljivija je što značajno olakšava i otkrivanje grešaka, ažuriranje i nadograđivanje. Dodavanje novih komponenti u sustav, kao i micanje nekih nepotrebnih, značajno je jednostavnije. Isto razmišljanje vrijedi i za različite programe koji pokretanjem postaju različiti zadaci.

Odvajanje nezavisnih poslova upravljanja u zasebne zadatke – koji time postaju nezavisne jedinice izvođenja – *dretve* zahtijeva *višezadaćni sustav* (engl. *multitasking*), tj. *višedretveni sustav*.

Višedretvenost omogućuje bolje iskorištenje računalnog sustava. U tom kontekstu bi se prednosti višedretvenosti mogle podijeliti na: ostvarivanje višezadaćnosti te mogućnosti paralelizacije jedne zadaće. Uobičajeno ponašanje pojedinih zadataka (dretvi) jest da nakon što nešto naprave trebaju čekati (na protok vremena, na drugi zadatak, na dovršetak operacije nad nekom napravom ili vanjskom jedinicom i slično). Dok je jedna dretva u stanju čekanja, procesor može izvoditi drugu dretvu koja ima nešto za napraviti (umjesto da procesor bude neiskorišten). Time se povećava iskoristivost računalnog sustava (on vrijedi više). Čak i dretve koje čekaju možda koriste elemente sustava. Primjerice, dretva koja želi nešto pročitati s diska mora čekati da disk dohvati njene podatke – ta je dretva "uposlila" disk; dretva koja šalje podatke preko mreže mora čekati da prethodno pripremljeni podaci budu isporučeni prije slanja novih – dretva je uposlila mrežnu karticu i slično. Paralelno se koriste različiti elementi sustava – opet se povećava iskorištenje sustava.

Ponekad, pojedini zadatak može sam iskoristiti više elemenata sustava njihovim paralelnim radom. Očito je da ako imamo više procesora[1] te ako se jedan zadatak može podijeliti na podzadatke, svaki bi se procesor mogao iskoristiti i time skratiti vrijeme izvođenja skupa podzadataka (dretvi) koji čine taj jedan zadatak (program). Također, ako ta jedna zadaća zahtijeva korištenje raznih elemenata sustava, podjela posla bi se mogla napraviti i prema tome te kao i kod višezadaćnosti bolje iskoristiti sustav (što bi u ovom jednozadaćnom primjeru značilo i brži dovršetak posla). Višedretvenost donosi i nove mogućnosti kod upravljanja i osmišljavanja

[1] U ovom se tekstu koriste pojmovi *procesor* i *procesorska jedinka*. Jedan procesor može imati ili samo jednu procesorsku jedinku ("stariji" procesori) ili ih može imati više, kao što su to *višestruki* (engl. *multicore*) i *mnogostruki* (engl. *manycore*) procesori. Više procesorskih jedinki imaju, dakle i višeprocesorski sustavi i sustavi s višestrukim/mnogostrukim procesorom.

arhitekture programske potpore.

Prednosti višedretvenosti se mogu podijeliti prema kriterijima:

- povećane učinkovitost korištenja sustava (sklopovlja) radi
  - višezadaćnosti;
  - paralelizacije intenzivnih računalnih problema koji se mogu rastaviti na (bar djelomično) neovisne dijelove kod višeprocesorskih sustava;
  - paralelnog korištenje različitih elemenata sustava (od strane različitih zadaća ili istog zadatka);
- jednostavnijeg ostvarenja sustava:
  - upravljanje različitim elementima sustava različitih svojstava i značaja (prioriteti, periodičko pokretanje) ostvaruje se zasebnim dretvama;
  - asinkrono upravljanje događajima/zahtjevima (primjerice početna dretva web poslužitelja za svaki novi zahtjev stvara zasebnu dretvu koja će posluživati zahtjev tako da početna dretva može i dalje nesmetano prihvaćati nove zahtjeve bez obzira o trajanju obrade već pristiglih);
  - jednostavnije ostvarenje složenog sustava korištenjem podjele i po dretvama (primjerice, jedna dretva brine o korisničkom sučelju, druga izvodi zadane proračune, treća prati stanje nekog procesa i slično).

Jedan od problema višedretvenosti jest u složenosti njegove ostvarivosti, tj. ugradnji podrške za višedretvenost u sustav. Drugi problem jest u korištenju višedretvenosti – dretve treba uskladiti, tj. sinkronizirati i osigurati mehanizme komunikacije i korištenja zajedničkih elemenata.

11.1.1. Načela višedretvenog rada

Osnovna ideja višedretvenosti jest u paralelnom radu više dretvi (stvarno paralelnom ili prividno paralelnom korištenjem načela podjele procesorskog vremena). Da bi se to dogodilo u sustavu mora biti više dretvi. One mogu biti dio istog posla ili pak svaka raditi svoj. Operacijski sustav treba omogućiti dinamičko pokretanje poslova – stvaranje novih dretvi, bilo preko sučelja prema korisniku, bilo preko sučelja prema programima koji sami stvaraju dodatne dretve.

Stanje nekog sustava određuje skup dretvi koje se u njemu nalaze i obavljaju svoje poslove. Neke od dretvi mogu biti u *stanju čekanja* ("blokirane"/"zaustavljene" dretve), tj. prije nego što nastave s radom moraju pričekati na neki "događaj" (akciju druge dretve, vanjske naprave ili protok vremena). Primjerice, dretva može čekati na naredbu korisnika, dohvat podatka s diska, istek prethodno zadanog vremenskog intervala i slično. Ostale dretve su "pripravne" i mogu se izvoditi na procesoru.

Korisnik "pokreće" poslove/programe pri čemu operacijski sustav stvara dretve koje ga obavljaju. S korisničke strane to je dovoljno; korisnik u daljem radu treba "samo" pratiti rad programa i po potrebi upravljati njegovim radom (unositi tražene podatke, pokretati željene operacije i slično). Po dovršetku posla ili po naredbi korisnika program se zaustavlja – dotične dretve se zaustavljaju i miču iz sustava.

Zadaća operacijskog sustava jest da upravlja dretvama, da im daje procesorsko vrijeme kad im je potrebno i kad je njihov red u odnosu na ostale dretve, da ih miče "na stranu" kada ne trebaju procesorsko vrijeme (kad čekaju na nešto), da ih stvara i dodaje u sustav na zahtjev drugih dretvi i korisnika te da ih miče iz sustava pri završetku njihova rada. Kako se sve navedene operacije zaista ostvaruju te kako su ostvarene u zadanom sustavu, prikazano je u nastavku.

11.1.2. Zadaće upravljanje dretvama

Sustav treba imati mogućnost upravljanja svim dretvama koje se nalaze u sustavu. U jednoprocesorskim sustavima u jednom trenutku može biti *aktivna* (izvoditi se na procesoru) samo jedna dretva. Sve ostale moraju čekati da ta završi ili ju operacijski sustav makne s procesora. Pojedina dretva može pripadati sustavu, obavljati potrebne operacije za sam sustav (obrada prekida, upravljanje ostalim zadacima i sredstvima sustava – *dretva sustava*, *jezgrena dretva*) ili dretva može pripadati nekom zadatku koji upravlja određenim vanjskim procesima ili pak dretva može pripadati programu koji je korisnik pokrenuo, a koji za njega obavlja korisne operacije (*korisnička dretva*).

Upravljanje dretvama treba omogućiti izvođenje svih dretvi u sustavu. Način i redoslijed izvođenja treba biti usklađen s važnošću posla koje dretve obavljaju – *prioritetom dretvi*. Korištenje sredstava sustava treba kontrolirati, ali i omogućiti njihovo korištenje od strane svih dretvi. U sredstva sustava spadaju: procesorsko vrijeme, spremnički prostor, UI naprave i ostala sredstva sustava (sinkronizacijski i drugi mehanizmi).

Upravljanje dretvama mora uzeti u obzir i posebnosti pojedinih dretvi. U sustavima za rad u stvarnom vremenu dretve često imaju vremenske okvire u kojima trebaju napraviti zadani posao. Takva ograničenja treba uzeti u obzir prilikom upravljanja dretvama.

11.1.3. Raspoređivanje dretvi

Problem raspoređivanja sredstava javlja se u gotovo svakom sustavu. Kako su gotovo svi sustavi upravljani računalima to postaje problem i u području računarstva. Raspoređivanje sredstava tako da se zadovolje sva ograničenja uz istovremeno postizanje očekivane učinkovitosti ili kvalitete, može biti vrlo složen problem. U nastavku se razmatra samo problem raspoređivanja dretvi u sustavu, odnosno raspoređivanje procesorskog vremena po dretvama sustava.

Raznolikost računalnih sustava zahtjeva razne metode raspoređivanja te se iz istog razloga nove metode neprestano istražuju i usavršavaju. Raspoređivanje u ugrađenim sustavima treba obaviti tako da sve dretve obave svoje (periodičke) poslove prije njihova trenutka krajnjeg završetka (prema slici 1.3.). Za različite probleme najčešće se koriste i različiti algoritmi raspoređivanja. Raspoređivač koji savršeno odgovara jednom tipu problema kod drugog može dati vrlo loše rezultate. Povećanje procesorske snage najčešće daje prave rezultate. Međutim, takvo rješenje poskupljuje gotovi proizvod te ga treba uzeti kao zadnje, ako se problem ne može riješiti drugim metodama. Ponekad ni zamjena jačim računalom nije dovoljna.

Najčešće korišteni algoritmi raspoređivanja dretvi u ugrađenim sustavima mogu se podijeliti na dvije skupine: *statički* i *dinamički*. Kod statičkih algoritama sustav se analizira prije samog rada te se unaprijed definira redoslijed izvođenja dretvi ili se dretvama statički pridijeli prioritet te ih onda sustav raspoređuje koristeći njihove prioritete. Dinamički algoritmi prate rad sustava i na osnovi njegova stanja dinamički određuju dretvu koja će se iduća izvoditi.

Problem raspoređivanja dretvi detaljnije je obrađen u drugim izvorima, primjerice [Budin, 2010], [Silbershatz, 2002], [Rajkumar, 1991], [SRSV, 2012]. Ovdje se koristi samo prioritetni raspoređivač koji za aktivnu dretvu uzima dretvu najvećeg prioriteta iz reda pripravnih i može se vrlo jednostavno ugraditi u sinkronizacijske funkcije. Algoritam raspoređivanja radi tako da se nakon svakog događaja koji uključuje funkcije sinkronizacije (jezgrene funkcije) prije samog izlaska iz funkcije od svih pripravnih dretvi za sljedeću aktivnu odabire ona s najvećim prioritetom.

Od ostalih načina raspoređivanja s mogućom primjenom u ugrađenim sustavima mogu se izdvojiti tri.

Algoritam mjere ponavljanja (engl. *rate monotonic scheduling – RMS* te *rate monotonic priority assignment – RMPA*) na osnovu učestalosti ponavljanja određuje prioritet dretvama te dalje

raspoređuje prema prioritetu.

Raspoređivanje prema trenucima krajnjih završetaka (engl. *earliest deadline first – EDF* te *deadline driven scheduling – DDS*) radi dinamički – u svakom trenutku promjene u sustavu razmatra sve zadatke i trenutke do kada oni trebaju obaviti svoj posao. Za aktivni zadatak odabire se onaj s najbližim (najskorijim) trenutkom krajnjeg završetka.

Raspoređivanje podjelom vremena (engl. *round robin – RR*) nastoji pravedno podijeliti procesorsko vrijeme među pripravnim dretvama po načelu "svakom malo" – svaka dretva dobije dio procesorskog vremena prije nego li se prekida i uzima iduća.

11.1.4. Ostvarivanje raspoređivanja dretvi

Raspoređivanje dretvi može biti napravljeno na nekoliko načina.

Najjednostavnije je *izravno raspoređivanje* kod kojeg trenutno aktivna dretva obavlja odabir iduće dretve i prebacuje joj kontrolu. Dretve su u takvom sustavu čvrsto povezane – svaka dretva mora znati za iduću u lancu ili čak sve ostale (kod složenijeg raspoređivanja). Logika raspoređivanja mora biti ugrađena u kôd svake dretve što otežava izgradnju dretvi. Kod ovog načina raspoređivanja, operacijski sustav (jezgra) nema kontrolu nad upravljanjem te mu taj podsustav i nije potreban – raspoređivanje je riješeno na *korisničkoj razini*, izvan jezgre pa se ovaj vid višedretvenosti ponekad naziva i *korisnička*, *programska* ili *kooperativna* višedretvenost.

Uobičajeno je da raspoređivanje ipak spada u domenu operacijskog sustava (njegove jezgre). Dretve i same mogu programski utjecati na način raspoređivanja, ali se samo raspoređivanje obavlja preko jezgrenih funkcija. Pri raspoređivanju razmatraju se samo dretve koje su spremne za izvođenje na procesoru – *pripravne dretve*, koje se nalaze u *redu pripravnih dretvi* (u nastavku se koristi termin *red pripravnih dretvi* iako je stvarna organizacija tih dretvi drugačija).

Osim u stanju pripravnosti (u redu pripravnih), dretva može biti i u nekoliko drugih stanja. Slika 11.1. prikazuje moguća stanja dretvi u sustavu.

Slika 11.1. Stanja dretvi u sustavu

U sustavima s jednom procesorskom jedinkom samo jedna dretva istovremeno može biti *aktivna*, u *aktivnom stanju* (u jednom trenutku samo se instrukcije jedne dretve mogu izvoditi). U sustavima s više procesorskih jedinki u istom trenutku može biti aktivno više dretvi. U ovoj skripti se ne razmatra ostvarenje operacijskog sustava za sustave s više procesorskih jedinki (Benu pretpostavlja da u sustavu postoji samo jedna procesorska jedinka).

U *pripravnom stanju* se nalaze dretve koje trebaju procesorsko vrijeme i koje jezgra po određenim kriterijima raspoređuje (prioriteti, FIFO, RR, ...). *Pripravne* dretve "čekaju" da se procesor oslobodi.

U nekom od *stanja čekanja* (*zaustavljenom*, *blokiranom* stanju) nalaze se dretve koje ne mogu

nastaviti s izvođenjem dok se neki uvjeti ne zadovolje (na primjer, sinkronizacijski uvjeti, kraj rada s UI napravama, oslobođenje nekog sredstva sustava, protok vremena, dolazak poruke/signala i slično). Za svako od stanja čekanja postoji zasebna lista ("red zaustavljenih dretvi").

U *pasivnom stanju* se nalaze dretve koje su završile sa svojim izvođenjem. Njihova se sredstva ponekad ne oslobađaju automatski, ako je primjerice potrebno njihovo izlazno stanje predati drugoj dretvi.

Načini raspoređivanja dretvi mogu se podijeliti u nekoliko osnovnih kategorija:

- raspoređivanje *prema redu prispijeća* (engl. *first come first served – FCFS* te češći naziv *first in first out – FIFO*)
- raspoređivanje *prema prioritetu*
- raspoređivanje *podjelom vremena – kružno raspoređivanje* (engl. *round robin – RR*).

U stvarnim se sustavima najčešće koriste kombinacije navedenih načina.

Najčešće ostvarenje raspoređivanja zasniva se na prioritetu kao osnovnom kriteriju – pri raspoređivanju odabire se pripravna dretva najvećeg prioriteta. Ako ima više takvih dretvi, moraju se koristiti i dodatni kriteriji kao što su red prispijeća i kružno posluživanje. Na primjer, ako se u sustavu, u trenutku raspoređivanja, nalaze pripravne dretve A, B i C s pridijeljenim im prioritetima $p_A = 5$, $p_B = 5$ te $p_C = 3$ (veći broj ovdje označava veći prioritet[2]), tada treba odabrati između dretvi A i B korištenjem drugog kriterija.

Podatkovna struktura koja podržava raspoređivanje prema prioritetima može se ostvariti uređenom listom u kojoj se nalaze dretve (njihovi opisnici) uređene prema prioritetima. Međutim, rad s uređenim listama je linearne složenosti te se u praksi takve liste rjeđe koriste (samo u sustavima s jako malo dretvi). Uobičajeno je da za svaki mogući prioritet – razinu prioriteta postoji po jedna lista (koja može biti i prazna). Slika 11.2. prikazuje moguću podatkovnu strukturu za organizaciju reda pripravnih dretvi.

Kada je lista ostvarena prema slici 11.2. ili slično, tada operacije nad svakom listom uključuju samo uzimanje prvog elementa te dodavanje na kraj liste. Obje su operacije složenosti $O(1)$ ako se za zaglavlje liste koriste kazaljke na prvi i zadnji element liste. U rijetkim će se prilikama ipak morati pretražiti lista (primjerice, kada se gasi neki proces treba obrisati sve njegove dretve, od kojih se neke možda nalaze i u stanju pripravnosti) – tada je složenost linearna.

Zamjena jedne dretve drugom zahtjeva dodatne operacije. Ako prva dretva koju se miče s procesora nije gotova već će kasnije nastaviti s radom, tada toj dretvi treba omogućiti kasniji nastavak s radom što znači osigurati joj isto stanje kao i u trenutku kad je zaustavljena. Sredstva koje dretva koristi nalaze se u radnom spremniku, ali i u registrima procesora. S obzirom na to da se podaci dretve u radnom spremniku ne mijenjaju radom druge dretve, ono što je potrebno sačuvati su registri procesora. Stoga se oni nazivaju i *kontekstom* dretve. Pri zamjeni jedne dretve drugom potrebno je najprije pohraniti kontekst prve, a potom obnoviti kontekst druge dretve.

Stanje stoga prije i poslije zamjene početne dretve drugom dretvom, kada se kontekst poharanjuje na stog dretvi, prikazano je na slici 11.3. Pri stvaranju nove dretve na stogu takve dretve treba pripremiti početni kontekst. Stoga je on detaljnije razmatran u nastavku.

[2]U različitim se sustavima različito poimaju relacije brojeva koje označavaju prioritet: najčešće veći broj označava veći prioritet (primjerice *Win32*, QNX Neutrino), ali negdje je obrnuto – manji broj označava veći prioritet (primjerice Windows CE). Benu koristi prvi pristup: veći broj označava veći prioritet.

Slika 11.2. Organizacija pripravnih dretvi u prioritetne razine – primjer stanja dretvi sustava

Slika 11.3. Stanje stoga prije i poslije zamjene dretvi

11.2. Višedretvenost ostvarena izvan jezgre

Podrška za višedretvenost mora se ugraditi u gotovo sve elemente sustava. Drugim riječima, višedretvenost je "skupa" za sustav. Zato se podrška za primitivne oblike višedretvenosti može ostvariti i izvan jezgre. Takav primjer višedretvenosti pokazan je u `Chapter_07_Threads/01_User_threads`. Sučelje višedretvene podrške ostvareno u `programs/user_threads/uthread.c` sastoji se od funkcija:

- za stvaranje nove dretve – `create_uthread(početna_funkcija, argument)`
- za kraj rada dretve – `uthread_exit()`
- za prepuštanja procesora drugim dretvama – `uthread_yield()`.

Operacija stvaranja nove dretve treba zauzeti spremnički prostor za opisnik dretve te za njen stog (instrukcije i podaci se već nalaze u spremniku). Kontekst dretve se može spremati na razna mjesta, u opisnik dretve, na stog, negdje drugdje. U ovom primjeru radi jednostavnosti odabran je stog.

Nova dretva će svoj rad započeti sa zadanom funkcijom kojoj treba predati jedan parametar. S obzirom na korišteni *gcc* i njegov način pozivanja funkcija (uobičajeni i za većinu ostalih prevoditelja), osnovni kontekst je oblikovan i proširen parametrom i povratnom adresom (prije samog konteksta). Naime, dretva svoj početak izvođenja treba započeti u zadanoj funkciji, a s obzirom na to da tada "očekuje" i "uobičajeni" stog, takvog treba i pripremiti.

U normalnom pozivu potprograma, prije instrukcije `call`, na stog se postavljaju parametri. Sama instrukcija `call` na stog dodatno stavlja povratnu adresu – adresu iduće instrukcije iza `call`. Na primjer, kada bi postojala funkcija:

```
int neka_funkcija ( int p1, int p2, int p3 )
{
    int x1, x2, x3;
    ...
```

koja se poziva s:

```
p = neka_funkcija ( a, b, c );
```

tada bi prevoditelj (primjerice *gcc*) taj poziv preveo u asembler (pojednostavljeno):

```
    push  c
    push  b
    push  a
    call  neka_funkcija
    add   $12, %esp
    mov   %eax, p
```

Prije poziva `call` na vrhu stoga se nalazi prvi parametar funkcije. Samim pozivom funkcije s `call` na stog se još stavlja povratna adresa (adresa iduće instrukcije po povratku iz potprograma, tj. adresa instrukcije `add $12, %esp`). Kôd početka funkcije očekuje ovakav stog te ako se on ne poziva na gornji način treba ga napraviti takvim, što je i učinjeno pri stvaranju početnog konteksta dretve. Lokalne varijable funkcije dodatno se postavljaju na stog – ali to radi kod u samoj funkciji. Prije povratka iz potprograma treba sve dodatno maknuti sa stoga tako da na vrhu bude povratna adresa.

[dodatno]

Najčešće se varijable ne koriste izravno, već preko registara. Također, umjesto instrukcije `push` može se koristiti i običan `mov` koji koristi kazaljku stoga. Za prethodni primjer poziva funkcije to može izgledati kao u nastavu.

```
    subl  $12, %esp
    movl  a, (%esp)
    movl  b, 4(%esp)
    movl  c, 8(%esp)
    call  neka_funkcija
    movl  %eax, p
    addl  $12, %esp
```

Razlog ovakva pristupa jest u optimiranju rada protočne strukture procesora obzirom da se na ovaj način kazaljka stoga ne mijenja i susjedne intrukcije se mogu paralelno izvoditi (ili bilo kojim redom), ovisno o dostupnim argumentima.

Pri prelasku na drugu dretvu, nakon spremanja konteksta prethodne, potrebno je u kazaljku stoga postaviti adresu vrha stoga te druge dretve te obnoviti kontekst s njega. Zato pri stvaranju nove dretve na stogu treba pripremiti početni kontekst dretve opisanim postupkom (za x86) funkcijom `arch_create_thread_context`. Izgled stoga nakon stvaranja početnog konteksta dretve prikazan je slikom 11.4.

Slika 11.4. Kontekst dretve i sadržaj stoga pri stvaranju nove dretve

Nakon što dretva obavi sve što joj je zadano, odnosno što se nalazi u njenoj početnoj funkciji, dretvu treba zaustaviti i maknuti iz sustava. S obzirom na to da sama dretva nije izravno pozvala svoju početnu funkciju, već njoj je okolina pripremljena da izgleda kao da jest, dretvi treba osigurati siguran završetak. Po obavljanju sveg posla u zadanoj funkciji, dretva će izaći iz početne funkcije na uobičajeni način – instrukcijom `ret`. Instrukcija `ret` će uzeti vrijednost s vrha stoga i staviti ju u programsko brojilo. Ako se tamo postavi adresa druge funkcije, dretva će po dovršetku početne započeti s tom drugom. Zato je i jedan od parametara funkcije `arch_create_thread_context` adresa "izlazne funkcije". Tim pristupom implicitno se postavlja funkcija `uthread_exit` kao završna operacija koju će dretva napraviti nakon povratka iz početne. U toj funkciji (`uthread_exit`) se dretva miče iz sustava. Isti je pristup korišten i u kasnijim izvedbama (kada se implicitno, preko strukture `prog_info_t` i njenog elementa `.exit`, funkcija `thread_exit` postavlja kao završna funkcija za svaku stvorenu dretvu).

Po stvaranju početnog konteksta i popunjavanju opisnika nove dretve, opisnik se smješta u red pripravnih dretvi. Iz reda pripravnih dretva će u nekom budućem trenutku otići u red aktivnih i nakon nekog vremena i možda dodatnih mijenjanja stanja iz pripravne u aktivnu i obratno, dretva će obaviti sve što joj je zadano u početnoj funkciji. Po završetku rada dretve, postupak njenog uklanjanja iz sustava je obrnut postupku dodavanja: opisnik te dretve se miče iz liste aktivne (dretva izravno poziva funkciju za dovršetak) te se spremnički prostor za opisnik i stog dretve oslobađaju.

U tijeku rada, kada je dretva aktivna, ona može izravno zatražiti prepuštanje procesora drugim dretvama. Operacija prepuštanja procesora drugoj dretvi podrazumijeva spremanje konteksta trenutne dretve, prebacivanje opisnika te dretve na kraj reda pripravnih dretvi, prebacivanje prve dretve iz reda pripravnih u red aktivne dretve te obnavljanje konteksta aktivne dretve. Navedene operacije napravljene su u funkciji `uthread_yield`. Pravo prebacivanje s jedne dretve na drugu obavlja se funkcijom `arch_switch_to_thread` definiranom u `arch/i386/context.c`. Kontekst dretve prema slici 11.4. definiran je u `arch/i386/context.h`.

Isječak kôda 11.1. Chapter_07_Threads/01_User_threads/arch/i386/context.h

```
13 typedef struct _arch_context_t_
14 {
15         int32    edi, esi, ebp, _esp, ebx, edx, ecx, eax;
16         uint32   eflags;
17         uint32   eip;
18 }
19 __attribute__((__packed__)) arch_context_t;
```

Oznaka `__attribute__((__packed__))` traži od prevoditelja da ne mijenja strukturu radi optimiranja već da ju izgradi upravo kako je i zadana (više o tome u C.2.3.). Kontekst se sprema na stog dretve pa operacija pohrane konteksta najprije sprema registar programskog brojila (`eip`), registar stanja (`eflags`) te konačno registre opće namjene (`eax`–`edi`). Te operacije su ostvarene korištenjem asemblera u funkciji `arch_switch_to_thread`.

Isječak kôda 11.2. Chapter_07_Threads/01_User_threads/arch/i386/context.c

```
38 /* switch from one thread to another */
39 void arch_switch_to_thread(context_t *from, context_t *to)
40 {
41         asm volatile (
42                 "cmpl   $0, %1          \n\t"   /* is "from" given? */
43                 "je     1f              \n\t"
44
45                 "pushl  $2f             \n\t"   /* EIP */
46                 "pushfl                 \n\t"   /* EFLAGS */
47                 "pushal                 \n\t"   /* all registers */
48                 "movl   %%esp, %0       \n\t"   /* save stack => from */
49
50         "1:      movl   %2, %%esp       \n\t"   /* restore stack <= to*/
51                 "popal                  \n\t"
52                 "popfl                  \n\t"
53                 "ret                    \n\t"
54
55         "2:      nop                    \n\t"
56
57                 : "=m" (from->context)  /* %0 */
58                 : "m" (from),           /* %1 */
59                   "m" (to->context)     /* %2 */
60         );
61 }
```

Ako `from` nije zadan (jednak je `NULL`) tada se ne sprema kontekst već se samo obnavlja kontekst zadane dretve (s adrese `to`).

Umjesto trenutne vrijednosti programskog brojila u kontekst dretve se sprema adresa instruk-

cije iza pohrane (i obnove konteksta). Oznaka `$2f` označava adresu iza labele `2:` (pretražujući od trenutne linije prema naprijed, `f`=*forward*), odnosno adresa instrukcije `nop` (brojčane labele tipa `1:`, `2:` i slično mogu se višestruko pojaviti u kôdovima prema *GCC* asemblerskim pravilima).

Zamjena jedne dretve drugom radi se dakle preko funkcije `arch_switch_to_thread`. Kada bi to bio jedini način zamjene, tada spremanje programskog brojila u kontekst ne bi bilo potrebno jer dretve ionako nastavljaju s radom na adresi `2:` nakon obnove konteksta. Međutim, kada to nije tako (npr. kada se dretva može promijeniti i u obradi prekida) tada postaje neophodno pohraniti i vrijednost programskog brojila. Izuzetak jest i prvo pokretanje dretve koja započinje s njenom početnom funkcijom, a ne kodom iz funkcije `arch_switch_to_thread`.

Nakon pohrane programskog brojila, registra stanja i ostalih registara na stog, adresa vrha stoga (`esp`) pohranjuje se u varijablu `from->context` (kontekst se sprema na stog te je dovoljno pamtiti vrh stoga). Kontekst druge dretve (`to`) nalazi se na njenom stogu te se najprije postavlja adresa stoga iz opisnika te dretve, tj. iz `to->context`, u registar `esp` prije obnavljanja konteksta: najprije registri opće namjene (`popal`) pa registar stanja (`popfl`) i na kraju programsko brojilo (instrukcijom `ret`).

Primjer iz `programs/user_threads/user_threads.c` prikazuje korištenje korisničkih dretvi. Iako je prikazano ostvarenje minimalno ono bi se moglo i u korisničoj razini proširiti i podrškom za prioritete, dodati mogućnost zaustavljanja dretvi, tj. redove za dretve u stanju čekanja (na sinkronizacijskom mehanizmu ili sredstvima sustava) i slično. Međutim, ostvarenje samo u korisničkoj razini bez podrške u jezgri ima i nekoliko ozbiljnih nedostataka. Prvi je u nemogućnosti povezivanja dretvi s UI napravama, tj. zaustavljanje dretvi dok čekaju na dovršetak UI operacija. Naime, s obzirom na to da dretve nisu povezane i upravljane u jezgri, nakon prekida se obnavlja kontekst dretve koja je prekidom bila prekinuta – nije moguće unutar obrade prekida zamijeniti aktivnu dretvu nekom drugom s obzirom na to da višedretvenost nije u jezgri. Taj problem onemogućava i ostvarivanje složenijih načina raspoređivanja, primjerice ostvarivanje raspoređivanja podjelom vremena (barem na uobičajeni, jednostavni način).

11.3. Višedretvenost ostvarena u jezgri

Prethodni problemi višedretvenosti ostvarene izvan jezgre rješavaju se ugrađivanjem višedretvenosti u jezgru što zahtjeva temeljitu prilagodbu jezgre (osnovni nedostatak ovog rješenja). Osnovna podrška višedretvenosti ostvarena u jezgri prikazana je u fazi `02_Threads`.

Ukratko, nakon stvaranja nove dretve (opisnik, početni kontekst na njenom stogu prema slici 11.4. i strukturi `arch_context_t`), opisnik dretve stavlja se u red pripravnih dretvi. Kad dođe njen red na izvođenje, ona se prebacuje u red aktivne dretve te se obnavlja kontekst te dretve (u funkciji `arch_switch_to_thread`) i ona započinje/nastavlja s radom.

S obzirom na to da je (u ovom inkrementu) razlika između korisničkog načina rada (rada u dretvi) i jezgrenog načina rada (rada unutar jezgrenih funkcije) jedino u tome što je prekid zabranjen u jezgrenim funkcijama, pozivi jezgri od strane dretvi su izravni pozivi (`sys__*` funkcije) kao i do sada, tj. jezgrene se funkcije obavljaju u kontekstu dretvi. To i nije problem jer se nakon poziva većinom sustav vraća u istu dretvu. Kada to nije slučaj, tj. kada poziv jezgre treba zamijeniti trenutno aktivnu dretvu nekom drugom, onda se to radi isključivo u funkciji `kthreads_schedule` (ostvarenoj u `kernel/sched.c`) na isti način kao što je to napravljeno za korisnički upravljane dretve, tj. pozivom funkcije `arch_switch_to_thread`.

Isti pristup koristi se i za pozive jezgri iz prekida naprava. Naime, iako se u prekidu kontekst tekuće (prekinute) dretve sprema u samom postupku prihvata prekida (`arch/i386/interrupt.S`), i dalje se ostaje u toj istoj dretvi, ali sada u njenom *prekidnom kontekstu* jer se i dalje koristi njen stog. U slučaju potrebe zamjene dretvi, taj se prekidni kontekst sprema opet kao kontekst te iste dretve (opet na stog iste dretve, ali ne prepisujući stari kontekst već povrh

njega). Pri povratku u tu istu dretvu najprije se obnavlja njen prekidni kontekst, a potom, "povratkom iz obrade prekida" i sam kontekst dretve te ona nastavlja s radom tamo gdje je stala u trenutku prekida naprave.

Navedeni načini promjene aktivne dretve u jezgrenim funkcijama i obradama prekida prikazani su slikom 11.5.

Slika 11.5. Zamjena dretve u prekidu i jezgrenoj funkciji

Prema slici 11.5. u postupku prihvata prekida (1) kontekst dretve se sprema na njen stog. U funkciji za obradu prekida – jezgrinoj funkciji, može se dogoditi da se nakon operacije jezgrine funkcije vraćamo i nastavljamo s prekinutom dretvom (6) ili ako treba neka druga dretva nastaviti s radom, onda se pozivom `kthreads_schedule` mijenja aktivna dretva – sprema se prekidni kontekst prethodne dretve i obnavlja (prekidni) kontekst neke druge dretve (koraci (2) i (3)). Nastavkom rada te druge dretve može se svašta promijeniti u sustavu. U nekom budućem trenutku ta će dretva (ili neka druga koja ju je zamijenila u međuvremenu) prekidom ili pozivom neke jezgrene funkcije (4) ponovno odabrati prvu dretvu (na slici označenu s "dretva 1") te s `kthreads_schedule` obnoviti njeni prekidni kontekst (5). Dovršetkom funkcije za obradu prekida, tj. povratkom iz prekida obnavlja se kontekst te dretve (6) i nastavlja se s njenim radom.

Pri izravnom pozivu jezgrene funkcije iz trenutne dretve, (1) predstavlja sam poziv, dok (6) predstavlja povratak iz te funkcije. Operacija (3) je zapravo povratak iz prekida. Operacija (4) se može dogoditi ili preko prekida ili izravnim pozivom jezgrene funkcije od strane druge dretve (koraci (1) i (2)).

Proširenje "podatkovne strukture jezgre" za potrebe upravljanja višedretvenošću sastoji se od *opisnika dretvi*, liste za razna stanja dretvi (aktivna, pripravne, zaustavljene, ...), proširenja drugih podsustava (upravljanje UI, alarmi), strukture za sučelja prema dretvama za upravljanje dretvama (dretve pozivaju funkcije za stvaranje novih dretvi, završetak dretvi, čekanje na kraj i ostale).

Osnovne operacije nad dretvama kao i potrebne strukture podatka smještene su u nekoliko datoteka:

- `kernel/thread.[hc]` – osnovne operacije, popis svih dretvi, kazaljka na aktivnu;

- `kernel/sched.[hc]` – upravljanje pripravnim dretvama (raspoređivanje), redovi za pripravne dretve;
- `kernel/pthread.[hc]` – proširenje operacija nad dretvama, POSIX sučelje.

Ugrađeno raspoređivanje dretvi kao prvi kriterij uzima prioritet, a kao drugi redoslijed prispijeća u red pripravnih dretvi, kao na primjeru na slici 11.2. Red za aktivnu dretvu je zapravo samo kazaljka na njen opisnik: `static kthread_t *active_thread` definirana je u `kernel/thread.c` gdje se nalaze i sve osnovne operacije nad opisnikom dretve. *Red pripravnih dretvi* `static sched_ready_t ready` definiran je u `kernel/sched.c` uz operacije raspoređivanja.

Red dretvi definiran je strukturom `kthread_q` čiji je jedini element opisnik liste (`list_t`).

Isječak kôda 11.3. Chapter_07_Threads/02_Threads/include/kernel/thread.h

```
9   /*! Thread queue (only structure required to be visible outside thread.c) */
10  typedef struct _kthread_q_
11  {
12          list_t  q;              /* queue implementation in list.h/list.c */
13          /* uint flags; */       /* various flags, e.g. sort order */
14  }
15  kthread_q;
```

U budućnosti bi se red dretvi mogao proširiti podacima koji bi definirali uređenje reda, da se osim po redu prispijeća, koji se sada koristi, može koristiti i neki drugi (primjerice prioritet).

U redu pripravnih dretvi za svaki mogući prioritet postoji zasebna lista pripravnih dretvi (element polja `rq` strukture `sched_ready_t` definirane u `kernel/sched.h`).

11.3.1. Opisnik dretve

Opisnik svake dretve mora sadržavati potrebne strukture podataka kako za ostvarenje listi u kojima se opisnik može nalaziti tako i za druge operacije nad dretvama. Opisnik `kthread_t` je definiran u `kernel/thread.h`.

Isječak kôda 11.4. Chapter_07_Threads/02_Threads/kernel/thread.h

```
98  /*! Thread descriptor */
99  struct _kthread_t_
100 {
101         id_t        id;
102                     /* thread id (number) */
103
104         int         state;
105                     /* thread state (active,ready,wait,susp.) */
106
107         int         flags;
108                     /* various flags (as detachable, cancelable) */
109
110         context_t   context;
111                     /* storage for thread context */
112
113         int         retval;
114                     /* return value from system call (when changed by others) */
115
116         int         errno;
117                     /* exit status of last system function call */
118
119         void       *exit_status;
120                     /* status with which thread exited (pointer!) */
121
122         void      (*cancel_suspend_handler)(kthread_t *, void *);
123         void       *cancel_suspend_param;
```

```
124         /* cancellation handler - when premature cancellation occurs while
125          * thread is suspended; used to perform required actions at cancellation
126          * event, e.g. when sleep is interrupted */
127
128         void      *pparam;
129                    /* temporary storage for one private parameter;
130                     * to be used only when thread is blocked to store single
131                     * parameter used by kernel only - not for private storage */
132
133         void      *stack;
134         uint       stack_size;
135                    /* stack address and size (for deallocation) */
136
137         int        sched_policy;
138                    /* scheduling policy */
139         int        sched_priority;
140                    /* priority - primary scheduling parameter */
141
142         kthread_q  *queue;
143                    /* in which queue thread is (if not active) */
144
145         kthread_q  join_queue;
146                    /* queue for threads waiting for this to end */
147
148         list_h     list;
149                    /* list element for "thread state" list */
150
151         list_h     all;
152                    /* list element for list of all threads */
153
154         int        ref_cnt;
155                    /* reference counter */
156 };
```

Identifikacijski broj se ne koristi izravno u jezgri (nema posebno važnu funkciju) već je prije zamišljen kao pomoć pri praćenju rada sustava (pri ispisu svih dretvi). Svaka dretva ima (dobiva) jedinstveni identifikacijski broj (funkcijom `k_new_id` u `kernel/memory.c`).

Stanje dretve može se pročitati iz elementa `.state` (`ACTIVE`, `READY`, `WAIT`, `PASSIVE`), ali i iz liste u kojoj se njen opisnik nalazi (`.queue`). Element `.list` služi za ostvarenje liste u kojoj se opisnik dretve nalazi, ovisno o stanju. Sve dretve, neovisno o stanju, nalaze se u dodatnoj listi (`all_threads`) te se za njeno ostvarenje koristi element `.all`.

Kontekst se trenutno sprema na stog dretve pa element opisnika dretve `.context` (sa svojim elementom `.context`) zapravo samo pokazuje na vrh stoga dretve (kada je ona zamijenjena drugom).

Stog dretve definiran je kazaljkom na početak spremnika zauzetog za stog te dretve (`.stack`) te njegovom veličinom (`.stack_size`). Ti su podaci potrebni pri završetku rada dretve, za oslobođenje tog dijela spremnika.

Status zadnje (jezgrene) funkcije – oznaka greške, sprema se u element `.errno`. Oznaka greške u višedretvenom sustavu vezana je uz pojedinu dretvu te je njen opisnik dretve logično mjesto pohrane oznake greške.

Dretve se u jezgrenim funkcijama zaustavljaju iz raznih razloga. Ponekad se uz propuštanje zaustavljene dretve želi promijeniti i vrijednost koju će ta dretva dobiti po nastavku rada. Zato je u opisnik dodan element `.retval` u koji se zapisuje povratna vrijednost u takvim okolnostima.

Dretva u stanju čekanja može biti propuštena i iz drugih razloga osim onih zbog kojih je bila zaustavljena. U takvim slučajevima može biti potrebno obaviti dodatne radnje prije povratka u tu dretvu. Tome služe elementi opisnika `cancel_suspend_handler` i `cancel_suspend_param`.

Parametri potrebni za raspoređivanje nalaze se u elementima `.sched_policy` i `.sched_priority`. Trenutno je podržano samo raspoređivanje prema prioritetu – dretva većeg prioriteta (veći broj u `.sched_priority`) istiskuje dretve manjeg prioriteta. U zadnjoj fazi ovog inkrementa su dodana još dva raspoređivača, ali s minimalnim utjecajem na osnovni raspoređivač. Dodatni raspoređivači su opisani u dodatku G.

Za čekanje na kraj druge dretve koristi se element (red) `.join_queue` (u koji se smještaju takve dretve, tj. njihovi opisnici) te element `.exit_status` za pohranu izlaznog statusa. Po završetku rada dretve, sve dretve koje čekaju u njenom redu `.join_queue` se propuštaju.

Opisnik dretve može još neko vrijeme ostati u sustavu i nakon završetka same dretve, dok se i zadnja dretva koja je čekala na kraj trenutne ne propusti i preuzme izlazni status. Da se opisnik prijevremeno ne bi obrisao koristi se element `.ref_cnt` koji broji poveznice iz drugih objekata na objekt opisnika dretvi. Dok je taj broj veći od nule opisnik se neće obrisati. Pri stvaranju i pokretanju dretve taj se broj postavlja na jedan. Pri završetku dretve broj se smanjuje za jedan. Za svaku dretvu koja se stavlja u red čekanja na završetak prve dretve, brojač se povećava za jedan, a svakim odlaskom dretvi iz reda, brojač se smanjuje za jedan. Ako je taj brojač jednak nuli u trenutku kada dretva završava (nakon smanjenja zbog toga) onda se opisnik dretve može obrisati. U protivnom on ostaje i briše se tek kada zadnja dretva napusti red `.join_queue` (briše se u toj funkciji).

11.3.2. POSIX sučelje za rad s dretvama

U osnovne operacije jezgre za rad s dretvama spadaju:

- stvaranje nove dretve
- završetak rada dretve
- čekanje na dovršetak druge dretve.

Stvaranje nove dretve pretpostavlja stvaranje okoline za novu dretvu. Okolina se sastoji od instrukcija, podataka, stoga i opisnika dretve. Instrukcije i podaci su već prisutni u radnom spremniku – učitani su zajedno sa sustavom. Za stog i opisnik treba zauzeti dijelove radnog spremnika te ih prilagoditi za početak dretve. Opisnik se potom stavlja u red pripravnih dretvi te se poziva raspoređivač.

Operacija završetka dretve je suprotna operaciji stvaranja: zauzeta sredstva dodijeljena dretvi se oslobađaju i brišu iz sustava i njegovih struktura podataka (listi).

Operacija čekanja na kraj druge dretve treba:

- zaustaviti pozivajuću dretvu, ako čekana još nije gotova
- propustiti pozivajuću dretvu, ako je čekana dretva već gotova i njen opisnik već obrisan (ne postoji) – pritom vratiti grešku ("dretva ne postoji")
- propustiti pozivajuću dretvu, ako je čekana već gotova, ali njen opisnik s izlaznim statusom je još pristan u sustavu (status se vraća pozivajućoj dretvi).

Navedene operacije se prema programima nude kroz POSIX sučelje za rad s dretvama. Iako su nazivi jezgrenih funkcija ostvarenih u `kernel/pthread.c` zbog slojevitosti (jezgra i sloj *api*) ponešto drukčiji (`sys__`*ime*), funkcionalnost je ista. Kratak opis funkcionalnosti slijedi u nastavku.

```
int pthread_create ( pthread_t *thread, pthread_attr_t *attr,
                     void *(*start_routine) (void *), void *arg );
```

Nova dretva stvara se pozivom `pthread_create`. Parametar `start_routine` definira početnu funkciju, a `arg` parametar za tu funkciju. S obzirom na to da je parametar kazaljka moguće je dretvi predati i više podataka ako se oni oblikuju u neku strukturu čija se adresa

može predati kao parametar. Opisnik dretve koji program (početna dretva) može koristiti (ne jezgren opisnik) sprema se na adresu `thread`. Dodatne atribute za dretvu, kao što su način raspoređivanja, prioritet, stog i slično, može se predati funkciji za stvaranje nove dretve preko parametra `attr`.

```
int pthread_exit ( void *retval );
```

Dretva svoj rad završava izlaskom iz početne funkcije ili pozivom `pthread_exit`. Kao izlazni status može navesti jednu vrijednost – adresu.

```
int pthread_join ( pthread_t thread, void **retval )
```

Čekanje na kraj druge dretve ostvaruje se funkcijom `pthread_join`. Prvi parametar funkcije `thread` mora biti opisnik dretve koju se čeka. Drugi parametar, ako nije `NULL`, pokazuje na adresu gdje će se spremiti izlazni status pohranjen s `pthread_exit` ili predan kao povratna vrijednost iz početne funkcije dretve (`return retval`).

Detaljniji opis ovih i drugih funkcija prema POSIX sučelju za rad s dretvama može se pronaći na [POSIX man].

11.3.3. Utjecaj višedretvenosti na ostale podsustave

Inicijalizacija podsustava za upravljanje dretvama (`kthreads_init`) mora se obaviti pri pokretanju sustava nakon inicijalizacije osnovnih podsustava (upravljanje napravama, spremnikom i ispisom). Inicijalizacija uključuje stvaranje listi za stanja dretvi (aktivno i pripravna), zauzimanje dijela spremnika za gomilu koja će se koristiti iz programa (odvojeno od jezgrene gomile) te popunjavanje strukture `prog_info_t` koja se koristi u programima. Dodatno, stvaraju se dvije dretve: *latentna dretva* i dretva programa s funkcijom definiranom `.init` elementom `prog_info_t` strukture (funkcija `prog_init`).

Isječak kôda 11.5. Chapter_07_Threads/02_Threads/kernel/thread.c

```
25 /*! initialize thread structures and create idle thread */
26 void kthreads_init()
27 {
28         int prio;
29
30         list_init(&all_threads);
31
32         active_thread = NULL;
33         ksched_init();
34
35         (void) kthread_create(idle_thread, NULL, 0, SCHED_FIFO, 0, NULL, 0);
36
37         /* initialize memory pool for threads */
38         pi.heap = kmalloc(PROG_HEAP_SIZE);
39         pi.heap_size = PROG_HEAP_SIZE;
40
41         prio = pi.prio;
42         if (!prio)
43                 prio = THREAD_DEF_PRIO;
44
45         (void) kthread_create(pi.init, NULL, 0, SCHED_FIFO, prio, NULL, 0);
46
47         kthreads_schedule();
48 }
```

Početna funkcija početne dretve (programa) `prog_init` inicijalizira standardne ulaze i izlaze, postavlja dinamičko upravljanje spremnikom za programe te pokreće "pravu" početnu funkciju definiranu `.entry` elementom, odnosno definiranu u `config.ini` varijablom `PROG_START_FUNC`. Korištena početna funkcija jest funkcija ljuske (`shell`), ali se isto tako može staviti bilo

koja druga.

Isječak kôda 11.6. Chapter_07_Threads/02_Threads/api/prog_info.c

```
/*! Initialize user process environment */
void prog_init(void *args)
{
        /* open stdin & stdout */
        stdio_init();

        /* initialize dynamic memory */
        pi.mpool = mem_init(pi.heap, pi.heap_size);

        /* call starting function */
        ((void (*)(void *)) pi.entry)(args);

        pthread_exit(NULL);
}
```

Višedretveni sustav treba projektirati imajući u vidu da različite dretve mogu trebati ista sredstva sustava, od spremničkih lokacija (dijeljenje strukture podataka) do sklopovskih sredstava (naprava), tj. treba razmišljati o "paralelnom radu sustava". Zato jezgra mora osigurati ispravno korištenje sredstava koji su kritični za rad sustava, kao što su to naprave i alarmi. Zato su ti podsustavi prvi prošireni.

Nadalje, jezgra mora osigurati mehanizme za programsko usklađivanje dretvi pri korištenju zajedničkih podataka programa nad kojima dretve obavljaju svoje operacije. Takve mehanizme nazivamo mehanizmima *sinkronizacije* i *komunikacije*. U idućim fazama sedmog inkrementa prikazani su sinkronizacijski mehanizmi semafora i monitora te komunikacijski mehanizmi razmjene poruka i signala.

Konačno, s obzirom na to da u nekom trenutku sve dretve mogu biti zaustavljene (zbog raznih razloga: UI naprave, sinkronizacijski mehanizmi), u sustav treba dodati i pomoćnu dretvu koja će "trošiti" procesor s obzirom na to da procesor mora izvoditi instrukcije. Dodana *latentna dretva* (engl. *idle thread*) ima najmanji mogući prioritet te se izvodi samo kada nema niti jedne druge dretve u redu pripravnih.

Podsustav za upravljanje vremenom (`kernel/time.c`) proširen je podrškom za višedretvenošću (u fazi `02_Threads`) tako da umjesto dotadašnjeg radnog čekanja i neproduktivnog trošenja procesorskog vremena, odgođene se dretve stavljaju u red odgođenih dretvi (nad objektom alarma koji je stvoren za tu odgodu) i prepuštaju procesor drugim dretvama (povećava se učinkovitost sustava).

Aktiviranje alarma u višedretvenom sustavu može se ostvariti na više načina:

- obavljajući zadanu funkciju aktivacije unutar obrade prekida izazvanog alarmom (u jezgrenom načinu rada uz zabranjeno prekidanje do završetka te funkcije)
- ubacivanje obrade funkcije kao prvog posla u dretvu koja je postavila taj alarm ili
- stvaranje nove dretve koja će obaviti zadanu funkciju.

U Benu su ostvareni svi navedeni načini: izravno pozivanje za alarme jezgre (interne alarme), ubacivanje obrade te stvaranje nove dretve za alarme stvorene u programima (korisničkim dretvama). Aktivacija stvaranjem nove dretve ostvarena je već u drugoj fazi, dok je ubacivanje obrade ostvareno s dodavanjem signala (u šestoj fazi). Kada se stvara nova dretva ona je po svojstvima slična dretvi koja je i stvorila alarm (isti prioritet, a kasnije i unutar istog procesa).

Primjer korištenja dretvi prikazan je u datoteci `programs/threads` čiji su najvažniji dijelovi navedeni u nastavku.

Posao dretve definira se njenom početnom funkcijom.

Isječak kôda 11.7. Chapter_07_Threads/02_Threads/programs/threads/threads.c

```
16 /* example threads */
17 static void *simple_thread(void *param)
18 {
19         int i, thr_no;
20
21         thr_no = (int) param;
22
23         printf("Thread %d starting\n", thr_no);
24         for (i = 1; i <= ITERS; i++)
25         {
26                 printf("Thread %d: iter %d\n", thr_no, i);
27                 nanosleep(&sleep, NULL);
28         }
29         printf("Thread %d exiting\n", thr_no);
30
31         return NULL;
32 }
```

Početna funckija dretve prima samo jedan parametar – kazaljku. Po potrebi se preko kazaljke dretvi može proslijediti i više podataka tako da kazaljka pokazuje na dio spremnika s tim podacima.

Isječak kôda 11.8. Chapter_07_Threads/02_Threads/programs/threads/threads.c

```
46         for (i = 0; i < THR_NUM; i++)
47                 if (pthread_create(&thread[i], NULL, simple_thread, (void *)i))
48                 {
49                         printf("Thread not created!\n");
50                         break;
51                 }
52
53         for (j = 0; j < i; j++)
54                 pthread_join(thread[j], NULL);
```

Iako je parametar dretve kazaljka, preko njega se mogu poslati i drugi jednostavni podaci poput cijelih brojeva (kazaljka je broj). Pritom je potrebno koristiti operatore pretvaranja vrijednosti (engl. *cast operator*).

Osim promjena u jezgri, podrška za višedretvenost je dodana i u sloju *api* preko nove datoteke `api/pthread.c` te dodacima i promjenama u ostalim elementima (`stdio.c`, `prog_info.c`, `time.c`).

11.4. Sinkronizacija i komunikacija

Korištenje višedretvenosti zahtjeva odgovarajuće mehanizme zaštite podataka od istovremenog korištenja te mehanizme komunikacije među dretvama te između dretvi i jezgre operacijskog sustava.

U prikazanom sustavu ostvareni su mehanizmi sinkronizacije semaforima i monitorima te mehanizmi komunikacije porukama i signalima.

11.4.1. Semafor

Jedan od jednostavnijih mehanizama sinkronizacije jest *semafor*. U svom uobičajenom obliku semafor se koristi za brojanje događaja, sredstava i slično. Sastoji se od jedne vrijednosti i reda za zaustavljene dretve (red semafora). Osnovne operacije nad semaforom su *ČekajSemafor* i *PostaviSemafor*.

Prva operacija, *ČekajSemafor*, pokušava zauzeti jedno sredstvo za pozivajuću dretvu, dok su-

protna operacija, *PostaviSemafor*, oslobađa jedno sredstvo (element polja, kritični odsječak i slično). Zauzeće i oslobađanje sredstava je zapravo samo logički povezano sa semaforom čije se operacije stavljaju na potrebna mjesta u programe. Same operacije se obavljaju samo nad objektom semafora (ne i sredstvima koja se štite). Operacije nad sredstvima slijede nakon zauzimanja semafora, a prije njegova otpuštanja.

Rezultati osnovnih operacija nad semaforom (*ČekajSemafor*, *PostaviSemafor*) ovise o stanju semafora. Uz svaki je semafor povezana varijabla koja označava trenutnu vrijednost semafora. Dodatno, s obzirom na to da semafor može biti i neprolazan, svaki semafor mora imati svoj red za zaustavljene dretve. Zajedno, varijable definiraju stanje semafora, koje može biti:

1. *prolazno*(vrijednost semafora veća od nule, red zaustavljenih dretvi prazan)
2. *neprolazno uz prazan red zaustavljenih dretvi*
3. *neprolazno uz neprazan red zaustavljenih dretvi* (barem jedna dretva se nalazi u redu).

Pri ostvarenju operacije *ČekajSemafor*, promatra se vrijednost semafora. Ako semafor ima vrijednost veću od nule, operacija će uspjeti (zauzeti sredstvo) i pritom će se vrijednost semafora smanjiti za jedan. Ako je vrijednost semafora već jednaka nuli, dretva koja poziva *ČekajSemafor* se zaustavlja, tj. njen opisnik se premješta u red semafora.

Operacija *PostaviSemafor* oslobađa jedno sredstvo na način da ako nema dretvi u redu semafora, povećava mu se vrijednost za jedan. U protivnom, prva dretva iz tog reda se propušta (njoj je "dodjeljeno" oslobođeno sredstvo).

Obje operacije nad semaforom moraju biti ostvarene kao nedjeljive (atomarne) operacije, tj. kao jezgrene funkcije te su u sljedećem pseudokôdu označene s `j_funkcija`.

```
j_funkcija ČekajSemafor ( id )
{
    ako ( Sem[id].v > 0 )
    {
        Sem[id].v = Sem[id].v - 1;
    }
    inače {
        stavi_u_red ( Sem[id].r, Aktivna_dretva );
        odaberi_aktivnu_dretvu(); //poziv raspoređivaču dretvi
    }
}

j_funkcija PostaviSemafor ( id )
{
    ako ( red Sem[id].r je prazan ) //ako nema dretvi u tom redu
    {
        Sem[id].v = Sem[id].v + 1;
    }
    inače {
        prva = uzmi_prvu_iz_reda ( Sem[id].r );
        stavi_u_red ( Pripravne_dretve, prva );
        odaberi_aktivnu_dretvu();
    }
}
```

Vrijednost semafora pohranjena je u element `.v` podatkovne strukture za semafor, dok element `.r` sadrži potrebne podatke za ostvarenje reda dretvi. Opisnici za semafore nalaze se u poretku `Sem[i]` iz kojeg se pojedini semafor dohvaća indeksom.

S obzirom na to da dretva propuštena u operaciji `PostaviSemafor` može imati veći prioritet od one koja poziva tu operaciju, prije povratka iz jezgrene funkcije treba obaviti raspoređivanje dretvi, tj. odabrati iduću aktivnu dretvu – ostaviti trenutnu ako je i dalje najvećeg prioriteta ili uzeti prvu iz reda pripravnih (upravo propuštenu).

Sukladno gornjim idejama, u fazi `04_Synchronization` u datoteci `kernel/pthread.c` ostvaren je semafor te je popraćen odgovarajućim funkcijama u `api/pthread.c` sa sučeljem prema POSIX normi.

POSIX sučelje za rad sa semaforima uključuje (pogledati `semaphore.h` za sve):

```
int sem_init ( sem_t *sem, int pshared, unsigned init_value );
int sem_post ( sem_t *sem );
int sem_wait ( sem_t *sem );
int sem_trywait (sem_t *sem );
int sem_timedwait ( sem_t *sem, const struct timespec *max_wait );
```

Sučelje `sem_trywait` je neblokirajuća inačica poziva za čekanje, koja će u slučaju da se semaforu vrijednost ne može smanjiti za jedan vratiti grešku kao povratnu vrijednost te pritom neće blokirati dretvu. Ograničeno blokiranje pruža sučelje `sem_timedwait`, koje će nakon isteka zadanog vremena dretvu odblokirati (ako se u međuvremenu nije odblokirala uobičajenim načinom – pozivom `sem_post` od strane neke druge dretve). Zadnje dvije operacije (`sem_trywait` i `sem_timedwait`) nisu prisutne u kôdu projekta, već su samo idejno opisane u dodatku E.1.

Primjer korištenja semafora naveden je u `programs/semaphores` na problemu proizvođača i potrošača koji komuniciraju preko ograničenog međuspremnika.

Semafor je vrlo jednostavan sinkronizacijski mehanizam i prikladan je za jednostavne sinkronizacije. U slučajevima složenijih sinkronizacija korištenje više semafora može dovesti do najgore situacije za dretve – do *potpunog zastoja*. Zato se u takvim složenijim sinkronizacijama preporuča korištenje drugog sinkronizacijskog mehanizma – *monitora*.

11.4.2. Monitor

Osnovna ideja mehanizma *monitora* jest da se složeni uvjeti provjeravaju u zaštićenom okruženju ("u monitoru") programskim ispitivanjem stanja sustava (varijabli koje opisuju to stanje). Ako je stanje "povoljno", dretva obavlja potrebne operacije ili zauzima potrebna sredstva – mijenja stanje sustava te potom napušta monitor. Ako stanje sustava "nije povoljno" za dretvu, ona ne može nastaviti s radom te se uvrštava u jedan od posebnih redova za zaustavljene dretve – u jedan od *redova uvjeta*. Time dretva ujedno i privremeno napušta monitor. Druge dretve mogu u monitoru promijeniti stanje sustava. Ako je neka od tih promjena "povoljna" za neku od zaustavljenih dretvi u redu uvjeta istog monitora, takvu dretvu treba propustiti, ali ne odmah u monitor jer bi tada u monitoru bile dvije dretve.

Rad u monitoru treba osigurati mehanizmom međusobnog isključivanja, tj. programsko ispitivanje treba se obavljati u *kritičnom odsječku* – samo se jedna dretva istovremeno može naći u monitoru.

Za ostvarenja mehanizma monitora potrebno je nekoliko jezgrenih funkcija. Za ulaz i izlaz potrebne su dvije. Za zaustavljanje dretve u nekom redu uvjeta (unutar monitora) potrebna je treća funkcija. Za propuštanje zaustavljenih dretvi trebaju dvije: jedna za propuštanje samo prve, a druga za propuštanje svih dretvi iz reda uvjeta. Potrebne jezgrene funkcije su dakle:

- `Zaključaj_monitor (m)`
- `Otključaj_monitor (m)`
- `Čekaj_u_redu_uvjeta (m, red)`
- `Propusti_iz_reda (red)`
- `Propusti_sve_iz_reda (red).`

Za svaki monitor je potrebna podatkovna struktura koja će imati element `.v` za odražavanje stanja monitora (pokazati je li neka dretva u monitoru ili nije), element `.r` za red dretvi

koje žele ući u monitor, ali trenutno ne mogu jer je monitor već zauzet (neka druga dretva je prethodno ušla u njega) te po jedan element `Red_uvjeta[j]` za svaki red uvjeta. Uz takvu strukturu podataka te polje opisnika monitora `Mon[]`, navedene se funkcije mogu prikazati pseudokôdovima:

```
j_funkcija Zaključaj_monitor ( m )
{
    ako ( Mon[m].v == 1 )
    {
        Mon[m].v = 0;
    }
    inače {
        stavi_u_red ( Mon[m].r, Aktivna_dretva );
        odaberi_aktivnu_dretvu();
    }
}

j_funkcija Otključaj_monitor ( m )
{
    ako ( red Mon[m].r je prazan )
    {
        Mon[m].v = 1;
    }
    inače {
        stavi_u_red ( Pripravne_dretve, uzmi_prvu_iz_reda (Mon[m].r) );
        odaberi_aktivnu_dretvu();
    }
}

j_funkcija Čekaj_u_redu_uvjeta ( m, red )
{
    pohrani_pripadajući_monitor ( Aktivna_dretva, m );
    stavi_u_red ( Red_uvjeta[red].r, Aktivna_dretva );

    ako ( red Mon[m].r je prazan )
    {
        Mon[m].v = 1;
    }
    inače {
        stavi_u_red ( Pripravne_dretve, uzmi_prvu_iz_reda (Mon[m].r) );
    }
    odaberi_aktivnu_dretvu();
}

j_funkcija Propusti_iz_reda ( red )
{
    ako ( red Red_uvjeta[red].r nije prazan )
    {
        prva = uzmi_prvu_iz_reda ( Red_uvjeta[red].r );
        m = dohvati_pripadajući_monitor (prva);

        ako ( Mon[m].v == 1 )
        {
            //dretva se propušta u monitor
            Mon[m].v = 0;
            stavi_u_red ( Pripravne_dretve, prva );
            odaberi_aktivnu_dretvu();
        }
        inače {
            //neka druga dretva je u monitoru,
            //treba pričekati da izađe

            stavi_u_red ( Mon[m].r, prva );
        }
    }
```

```
}

j_funkcija Propusti_sve_iz_reda ( red )
{
    dok ( red Red_uvjeta[red].r nije prazan )
    {
        prva = uzmi_prvu_iz_reda ( Red_uvjeta[red].r );
        m = dohvati_pripadajući_monitor (prva);

        ako ( Mon[m].v == 1 )
        {
            //dretva se propušta u monitor
            Mon[m].v = 0;
            stavi_u_red ( Pripravne_dretve, prva );
            odaberi_aktivnu_dretvu();
        }
        inače {
            //neka druga dretva je u monitoru,
            //treba pričekati da izađe

            stavi_u_red ( Mon[m].r, prva );
        }
    }
}
```

Razlika između zadnjih dviju funkcija je u početnom `ako` — `dok` dijelu. Ako se pri propuštanju dretve iz reda uvjeta neka druga dretva nalazi u monitoru (primjerice ona koja zove funkciju `propusti*`), "propuštena" dretva se ne propusti u monitor već se stavlja u red za ulaz u monitor. Inače se propušta u monitor, tj. stavlja se u red pripravnih dretvi.

Treba primijetiti da se funkcije `Propusti_iz_reda` i `Propusti_sve_iz_reda` mogu pozvati i izvan monitora (za razliku od `Čekaj_u_redu_uvjeta` i `Otljučaj_monitor`). Također, dretve mogu biti zaustavljene na redu uvjeta unutar različitih monitora (vrlo su rijetke okolnosti kada tako nešto ima smisla).

Ostvarenje monitora u kôdu odgovara prikazanom rješenju u pseudokôdu, uz imena sučelja prema POSIX normi: `pthread_mutex*`, `pthread_cond*`.

Za ostvarenje monitora dodatno potrebni mehanizam jezgre jest pohrana dodatnih podataka pri zaustavljanju dretve (za poziv `Čekaj_u_redu_uvjeta`). Opisnik dretve je stoga proširen kazaljkom `.pparam` u koju se pohranjuje kazaljka na monitor unutar kojeg je dretva zaustavljena u redu uvjeta, a koji je potreban pri propuštanju dretve iz reda uvjeta.

Kao i za semafor, tako i za monitor za posebna okruženja mogu biti potrebna slična proširenja: vremenski ograničeno zaustavljanje na ulaz u monitor te operacija ulaza u monitor koja ne zaustavlja dretvu ni kada se ne može ući u monitor (kada je neka druga dretva već u monitoru). Dodatno, za monitor se može definirati ponašanje u slučaju prvotno neočekivanih načina korištenja. Moguća proširenja su opisana u dodatku E.

Osnovna i proširena POSIX sučelja za ostvarenje monitora su:

```
int pthread_mutex_init   ( pthread_mutex_t *mutex,
                           const pthread_mutexattr_t *attr );
int pthread_mutex_lock   ( pthread_mutex_t *mutex );
int pthread_mutex_unlock ( pthread_mutex_t *mutex );
int pthread_cond_init    ( pthread_cond_t *cond,
                           const pthread_condattr_t *attr );
int pthread_cond_wait    ( pthread_cond_t *cond,
                           pthread_mutex_t *mutex );
int pthread_cond_signal  ( pthread_cond_t *cond );
int pthread_cond_broadcast  ( pthread_cond_t *cond );
int pthread_mutex_trylock   ( pthread_mutex_t *mutex );
int pthread_mutex_timedlock ( pthread_mutex_t *mutex,
```

```
                                const struct timespec *abstime );
int pthread_cond_timedwait  ( pthread_cond_t *cond,
                                pthread_mutex_t *mutex,
                                const struct timespec *abstime );
```

Operacije `*try*` i `*timed*` nisu ostvarene u Benu.

Korištenje monitora prikazano je na problemu *pet filozofa* u `programs/monitors`.

11.4.3. Poruke

Komunikacija među dretvama može se ostvariti i "ručno" korištenjem zajedničkog spremničkog prostora i nekoliko semafora, kao što je to prikazano u primjeru proizvođača i potrošača. Međutim, ako su podaci koji se razmjenjuju kratki (primjerice do stotinjak okteta) tada je navedeni pristup zahtjevan za sustav jer zahtjeva tri do četiri semafora i nekoliko poziva jezgrenih funkcija za sinkronizaciju pri samo jednoj razmjeni podataka. S obzirom na to da je potreba za razmjenom kratkih informacija između dretvi učestala, u sustave (operacijske sustave) se dodaje mehanizam koji to izravno podržava – *mehanizam razmjena poruka*.

Poruka je kratka informacija zadana svojom *veličinom* i *oznakom*. Oznaka nije obavezna, ali može poslužiti za odabir koja će se poruka uzeti iz reda.

Poruka se šalje u *red poruka*. Red poruka može biti uređen ili prema redu prispijeća (najčešći način organizacije) ili na neki drugi način (prema oznakama). Red poruka može pripadati nekoj dretvi ili može biti nezavisan (globalan). U sustavima za rad u stvarnom vremenu je uobičajeno da se uz svaku dretvu veže i jedan red poruka za poruke upućene izravno toj dretvi.

Sučelje za ostvarenje reda poruka osim sučelja za dohvat i brisanje reda poruka, mora imati dvije operacije:

- *PošaljiPoruku (red, poruka, duljina, tip)* te
- *PročitajPoruku (red, poruka, duljina, tip)*.

Čitanje poruke miče poruku iz reda, tj. jedna se poruka može samo jednom pročitati od strane jedne dretve. Ako je zadan tip poruke, pri čitanju se red pretražuje dok se poruka zadanog tipa ne pronađe – ako je nema vraća se greška ili se dretva zaustavlja (ovisno o zastavicama s kojima je red otvoren). Slika 11.6. prikazuje idejni prikaz reda poruka i način organizacije reda po redu prispijeća.

Slika 11.6. Idejni prikaz reda poruka

Operacije stavljanja poruke u red poruka i čitanja poruke iz reda poruka bit će donekle različite ovisno o tome ima li dretvi koje čekaju na poruku ili prazno mjesto u redu (da mogu staviti svoju poruku). Prilikom slanja poruke:

- kada je red pun – dretva se zaustavlja

- kada ima mjesta u redu poruka – poruka se stavlja u red poruka (dodaje u listu za poruke) te ako ima dretvi koje čekaju na poruku prva dretva iz tog reda se propušta (ta će dretva ponovno pokušati kad postane aktivna).

Prilikom čitanja poruke:

- kada je red prazan – dretva se zaustavlja
- kada ima poruka u redu – uzima se prva poruka te ako ima dretvi koje čekaju na prazno mjesto u redu prva dretva iz tog reda se propušta (ta će dretva ponovno pokušati kad postane aktivna).

Ostvarene funkcije imaju sučelje prema POSIX normi:

```
mqd_t mq_open ( char *name, int oflag,
                mode_t mode, struct mq_attr *attr );

int mq_send ( mqd_t mqdes, char *msg_ptr,
              size_t msg_len, uint msg_prio );

ssize_t mq_receive ( mqd_t mqdes, char *msg_ptr,
                     size_t msg_len, uint *msg_prio );
```

Više o parametrima i načinu korištenja potražiti u [POSIX man] te pogledati primjer `programs/messages`.

11.5. Signali

Do sada predstavljenim mehanizmima jezgre mogu se ostvariti mnoge operacije i među dretvama i za upravljanje pristupom napravama (preko jezgre). Problem ostaju sporadični i asinkroni događaji koji zahtjevaju posebnu reakciju dretve. Ako dretva programski "čeka" na takav događaj (primjerice zaustavljena je na napravi koja izaziva taj događaj) ona će ga moći odmah po pojavi i obraditi. Ali što ako ona radi nešto drugo u trenutku kad se takav događaj pojavi?

Sličan problem na razini procesora i UI naprava riješen je mehanizmom *prekida*, gdje se trenutna dretva prekida radi obrade prekida. Događaj na višoj razini koji se tiče dretve treba riješiti na sličan način: privremeno prekinuti dretvu radi obrade događaja. Detekcija "događaja" obavlja se u jezgrenim funkcijama kao reakcija na događaj UI naprave ili druge dretve (preko jezgre) te je zadaća jezgre da to i "dojavi" dretvi koje se taj događaj tiče.

Uobičajeni mehanizam za sporadične događaje su *signali* ostvareni na razini operacijskog sustava. Signali su slični prekidima na razini procesora uz razliku da signale (izravno ili neizravno) šalju i jezgra operacijskog sustava i dretve, a "primaju" ih dretve. Ponašanje dretve za pojedini signal se u većini sustava može prilagoditi tako da ona:

- prihvaća signal vlastitom (programski definiranom) funkcijom
- prihvaća signal na uobičajeni način (pretpostavljenom, ugrađenom funkcijom)
- ignorira signal ili
- zadržava signal – signal se ne odbacuje, ali se trenutno niti ne prihvaća – signal ostaje u sustavu dok se ponašanje dretve za njega ne promijeni.

Očekivano ponašanje dretve po primitku signala je:

- prekinuti s trenutnim izvođenjem instrukcija
- zabraniti daljnje prekidanje s tim signalom
- pohraniti kontekst trenutnog posla na stog
- skočiti u funkciju za obradu signala

- *obraditi signal*
- vratiti se iz obrade signala: obnoviti kontekst sa stoga i ponovno dozvoliti prekidanje istim signalom.

Većina sustava nastoji prihvat signala obaviti prema gornjem postupku. Međutim, navedeni postupak ima nekoliko problema. Što ako je dretva kojoj se signal šalje privremeno zaustavljena (primjerice u redu semafora ili čeka na dohvat podataka s naprave)? Na prvi pogled moglo bi se opet isto napraviti: obraditi signal i vratiti dretvu u prijašnje stanje. Jedan problem jest u tome što se s redom u kojem je dretva bila u međuvremenu može svašta dogoditi. Drugi problem je što se u obradi signala svašta može napraviti pa i ponovno zaustaviti dretvu.

UNIX i slični sustavi definiraju što će se dogoditi u slučaju primitka signala dok je dretva bila u nekoj jezgrenoj funkciji i to *za svaku jezgrenu funkciju zasebno* (za funkcije koje reagiraju na signal). Uobičajeno je da se dretva vraća iz tih funkcija s greškom kao povratnom vrijednošću te postavljanjem oznake greške u *EINTR* – "jezgrena funkcija prekinuta signalom" (engl. *a signal interrupted the call*).

Signali ostvareni u Benu podržavaju dio funkcionalnosti koje POSIX definira. Može se postaviti da se signal obrađuje postojećom dretvom tako da se privremeno prekida njen rad, stvara novo "stanje" dretve u kojem se signal obrađuje (koristi se i zasebni stog).

Ako je dretva bila u stanju čekanja u trenutku primitka signala, ona će najprije obraditi primljeni signal (kako je prethodno definirano). Potom će biti izbačena iz jezgrene funkcije u kojoj je prethodno bila zaustavljena uz dojavu greške i postavljene oznake greške u *EINTR*.

Uobičajeno ostvarenje signala pretpostavlja da uz sam signal, tj. identifikaciju koji se signal pojavio, nema drugih informacija koje se šalju dretvi. U nekim proširenjima (primjerice *POSIX RT*) se uz signal može vezati i kratka informacija. S obzirom na to da se ovdje razmatraju i SRSV-ovi, prikazani sustav podržava slanje dodatne informacije uz sam signal. Zapravo su samo takva sučelja i ostvarena (nisu podržani signali bez dodatne informacije).

Dodatkom podrške za signale proširen je i podsustav za upravljanje vremenom, s obzirom na to da alarmi u svojoj aktivaciji sada mogu slati i signale te da signali mogu prekidati odgođene dretve. Dijelovi opisnika dretve i neke funkcije za upravljanje dretvama tek se sada koriste u prethodno samo zamišljenom scenariju. Dijelovi opisnika koji se mogu mjenjati u obradi signala su izdvojeni iz opisnika dretve, odnosno definirana je struktura koja opisuje *stanje dretve* i koja se stvara i koristi svaki put kada se započne obrada signala. Kada obrada signala završi stanje se briše i obnavlja se prethodno stanje dretve.

Promjene izazvane dodavanjem signala su znatne te se neće detaljno opisivati. One se mogu vidjeti usporedbom faze `06_Signals` s prethodnom (u izvornim kôdovima).

Primjer korištenja signala naveden je u `programs/signals`. Početna dretva definira funkciju `sig_handler` za prihvat signala `SIGUSR1` te potom stvara alarm koji će joj svakih 6 sekundi slati taj signal. Nadalje, prikaz korištenja funkcije `sigwaitinfo` ostvaren je stvaranjem dodatne dretve koja najprije zabranjuje prihvat svih signala te ih potom prihvaća funkcijom `sigwaitinfo`.

Postavljanje akcije na signal, stvaranje alarma koji šalje signal, stvaranje nove dretve, obavljanje nekog posla koji traje 10 sekundi, brisanje alarma, slanje signala dretvi, čekanje na dovršetak dretve – sve to prikazuje idući kôd koji obavlja početna dretva u svom "normalnom" radu (prije obrade signala).

Isječak kôda 11.9. Chapter_07_Threads/06_Signals/programs/signals/signals.c

```
72          /* signal on timer activation */
73          evp.sigev_notify = SIGEV_SIGNAL;
74          evp.sigev_signo = SIGUSR1;
75
76          act.sa_sigaction = sig_handler;
```

```
77          act.sa_flags = SA_SIGINFO;
78          sigemptyset(&act.sa_mask);
79          sigaction(SIGUSR1, &act, NULL);
80
81          /* timer1 */
82          t1.it_interval.tv_sec = 6;
83          t1.it_interval.tv_nsec = 0;
84          t1.it_value.tv_sec = 6;
85          t1.it_value.tv_nsec = 0;
86          evp.sigev_value.sival_int = SIGUSR1;
87          timer_create(CLOCK_REALTIME, &evp, &timer1);
88          timer_settime(&timer1, 0, &t1, NULL);
89
90          if (pthread_create(&thread, NULL, signal_waiting_thread, NULL))
91                  printf("Thread not created!\n");
92
93          t.tv_sec = 1;
94          t.tv_nsec = 0;
95          sem_init(&sem, 0, 3);
96          for (i = 0; i < 10; i++)
97          {
98                  printf("In main thread(%d)\n", i);
99                  if (i < 5 && sem_wait(&sem) == EXIT_FAILURE)
100                 {
101                         int errno = get_errno();
102                         printf("sem_wait interrupted, errno=%d\n", errno);
103                         continue;
104                 }
105
106                 if (clock_nanosleep(CLOCK_REALTIME,0,&t,NULL) == EXIT_FAILURE)
107                 {
108                         int errno = get_errno();
109                         printf("Interrupted, errno=%d\n", errno);
110                 }
111         }
112
113         timer_delete(&timer1);
114
115         /* send signal to waiting thread */
116         sigval.sival_int = SIGUSR2;
117
118         /* send signal */
119         sigqueue(thread, SIGUSR2, sigval);
120
121         pthread_join(thread, NULL);
```

Funkcija za obradu signala dohvaća podatke o signalu, ispisuje poruku te simulira obradu.

Isječak kôda 11.10. Chapter_07_Threads/06_Signals/programs/signals/signals.c

```
13 static void sig_handler(siginfo_t *siginfo)
14 {
15         int num;
16         timespec_t t, t1;
17         int i;
18
19         num = siginfo->si_value.sival_int;
20         clock_gettime(CLOCK_REALTIME, &t);
21         time_sub(&t, &t0);
22
23         printf("[%d:%d] Signal %d\n",
24                 t.tv_sec, t.tv_nsec/1000000, num);
25
26         t1.tv_sec = 1;
27         t1.tv_nsec = 0;
28         for (i = 1; i < 4; i++)
```

```
29          {
30                  printf("In signal handler(%d)\n", i);
31                  clock_nanosleep(CLOCK_REALTIME, 0, &t1, NULL);
32          }
33  }
```

Dodatna dretva (stvorena od početne) na svom početku maskira sve signale (zabranjuje njihov prihvat), ali ih potom "ručno" dohvaća, tj. čeka na njih.

Isječak kôda 11.11. Chapter_07_Threads/06_Signals/programs/signals/signals.c

```
35  static void *signal_waiting_thread(void *param)
36  {
37          sigset_t set;
38          siginfo_t info;
39
40          sigfillset(&set);
41          pthread_sigmask(SIG_BLOCK, &set, NULL);
42
43          printf("Signal waiting thread started\n");
44          sigwaitinfo(&set, &info);
45          printf("Signal waiting thread got signal:"
46                   "num=%d, code=%d, errno=%d, si_value=%d\n",
47                   info.si_signo, info.si_code, info.si_errno,
48                   info.si_value.sival_int);
49
50          return NULL;
51  }
```

Prije korištenja signala u nekom sustavu potrebno je detaljno proučiti dokumentaciju tog sustava. Naime, iako mnogi sustavi deklariraju da podržavaju signale preko POSIX sučelja, neke operacije mogu biti malo drukčije ili nisu podržane u potpunosti (kao što je to primjerice sam Benu).

Sučelje (POSIX i starije) za signale uglavnom spominje procese, a ne dretve. Međutim, za očekivati je da će se u skoroj budućnosti više pažnje posvetiti dretvama i da će i ona sučelja koja su definirana standardom biti sve više ostvarivana u operacijskim sustavima. Osnovna sučelja za upravljanje signalima navedena su u nastavku (nisu sva ostvarena u Benu).

Postavljanje reakcije na signal može se napraviti funkcijom:

```
int sigaction ( int sig,
                const struct sigaction *act,
                struct sigaction *oact);
```

Slanje signala drugoj dretvi obavlja se funkcijama:

```
int kill ( pid_t pid, int sig );
int pthread_kill ( pthread_t thread, int sig );
int raise ( int sig );
int sigqueue ( pid_t pid, int signo, const union sigval value );
```

Poziv `sigqueue` jedini omogućava slanje signala uz koje ide i dodatna informacija. Za prihvat takvih signala funkcija za obradu mora imati dodatne parametre, npr. prema:

```
void obrada_signala ( int signum, siginfo_t *info, void *context );
```

ili takve signale dohvaćati funkcijom:

```
int sigwaitinfo ( const sigset_t *set, siginfo_t *info);
```

Mnogi mehanizmi se zasnivaju na signalima. Primjerice, signali se koriste za ostvarivanje odgode izvođenja, periodičko pokretanje, dojavu promjena na ulazno-izlaznim napravama itd.

Dodatni primjer sa signalima nalazi se u [Jelenković, 2010].

Pitanja za vježbu 11

1. Podrška za višedretvenost iziskuje znatne dodatne operacije jezgre. Kada podršku za višedretvenost ima smisla ugrađivati u sustav, a kada ne?
2. U kojim se stanjima može naći dretva u računalnom sustavu (u operacijskom sustavu)?
3. Koje se dretve (u kojim stanjima) razmatraju pri raspoređivanju dretvi? Opisati tipičnu podatkovnu strukturu u kojoj su takve dretve (njihovi opisnici).
4. Što je to opisnik dretve i koji se podaci trebaju naći u njemu?
5. Zašto treba izbjegavati uređene liste za opisnike dretvi (i druge svrhe)?
6. Upravljanje dretvama zahtjeva poznavanje operacija sa stogom na niskoj razini, kao i pozivanje potprograma (i povratak iz njega). Koji je najčešći način pozivanja potprograma (kamo se stavljaju parametri, kako se sprema povratna adresa, kamo se sprema povratna vrijednost)?
7. Kako se jedna dretva (aktivna) zamjenjuje drugom u postupku raspoređivanja? Što je to *zamjena konteksta* i kako se izvodi?
8. Navesti i opisati osnovne načine (principe) raspoređivanja dretvi.
9. Ako je neka dretva stanju čekanja (zaustavljena/blokirana), kako i kada će ta dretva nastaviti s radom? Koji razlozi mogu uzrokovati zaustavljanje dretve?
10. Najčešće korišteni sinkronizacijski mehanizmi su semafori i monitori. Koji su osnovni principi rada tih mehanizama? Koje su osnovne operacije tih mehanizama (osnovne funkcije)? Koja su moguća proširenja tih operacija?
11. Navedite nekoliko ostalih sinkronizacijskih mehanizama (osim semafora i monitora) koji se često koriste. Opišite osnovne principe tih mehanizama.
12. Slično prekidima na "niskoj" razini – razini procesora, događaji (signali) se koriste na razini operacijskog sustava. Koja su uobičajena ponašanja koja dretva može definirati za svaki signal koji je njoj upućen?
13. Signal se dretvi može poslati u "neočekivanom" trenutku. Koji problemi mogu nastati zbog toga (kako/kada prihvatiti signal)?
14. Komunikacija porukama je često jedino komunikacijsko sredstvo ugrađenim sustavima. Kako ga ostvariti? Koji su pritom problemi? Zašto je ta komunikacija tako često korištena upravo u tim sustavima?
15. Opisati POSIX sučelje za upravljanje dretvama, sinkronizaciju i komunikaciju.

12. Procesi

Podrška za višedretvenost prikazana u 11. poglavlju može poslužiti za mnoge sustave. Ipak, za uporabu u složenijim i zahtjevnijim sustavima navedeno rješenje ima i nedostatke. Jedan od njih jest u nemogućnosti bolje zaštite jezgre od neispravnih programa. Naime, s obzirom na to da se i dretve (programi) i jezgrene funkcije izvode u istom, privilegiranom načinu rada i da dijele cjelokupni adresni prostor (spremnik i UI naprave), ispad ili greška bilo koje dretve može narušiti rad i ostalih dretvi i sustava u cjelini. Primjerice, zbog greške u kazaljki mogu se prebrisati neki podaci sustava (primjerice opisnici druge dretve) iz kojih se sustav neće moći oporaviti.

U složenijim jezgrama operacijskih sustava problem zaštite se rješava korištenjem sklopovske potpore za odvajanje jezgre i svakog programa u zasebni spremnički prostor. Također, korištenjem mogućnosti procesora, *programi*, koji time postaju *procesi*, se izvode u *korisničkom načinu rada*, dok se jezgrene funkcije izvode u *jezgrenom* ili *privilegiranom načinu rada*.

Proces je dakle okolina u kojoj se izvodi program. Okolina je definirana dijelovima spremničkog prostora koji su na raspolaganju programu. U izvođenju, početna dretva koja izvodi instrukcije programa može stvoriti i dodatne dretve (sučeljem jezgre) tako da se unutar istog procesa može istovremeno nalaziti i nekoliko dretvi. Jedan proces od drugog procesa zaštićen je ograđenim spremničkim prostorima, tj. dretva jednog procesa ne može pristupiti spremničkom prostoru drugog procesa (ne bez posebnih zahtjeva jezgri). Također, dretve nekog procesa ne mogu izravno pristupiti podacima jezgre jer su podaci jezgre izvan adresnog prostora procesa (nisu mu dohvatljivi).

Izvođenjem u korisničkom načinu rada procesora dretvama se onemogućava pokretanje privilegiranih instrukcija koje mogu narušiti stabilnost sustava. Primjerice, korisničke dretve tako ne mogu zabraniti prekidanje, ne mogu mijenjati sadržaje registara koji upravljaju prihvatom prekida, registara koji definiraju upravljanje spremnikom i slično.

Kako ostvariti navedene zaštite? U nastavku su prikazani neki pristupi na primjeru arhitekture x86.

12.1. Načini rada procesora x86

Porodica procesora IA-32 (x86) podržava nekoliko načina rada (izvorni nazivi: *protected mode*, *real-address mode*, *system management mode*, *virtual-8086 mode*, *IA-32e mode*). U ovom projektu (Benu) koristi se samo jedan: *zaštićeni način rada* (engl. *protected mode*), u kojem se uobičajeno koristi i u kojem se mogu iskoristiti sve njegove mogućnosti. U tom načinu rada postoje četiri stanja s različitom *razinom privilegija*, takozvanim *prstenovima* od 0 do 3. Prsten 0 označava stanje s najvećom razinom privilegija, dok prsten 3 stanje s najmanjom razinom privilegija. U nastavku će se razmatrati i koristiti samo prsteni 0 i 3 te će se stanje u prstenu 0 nazivati *privilegirani način rada procesora*, tj. *jezgren način rada*, a stanje u prstenu 3 će se nazivati *neprivilegirani način rada* ili *korisnički način rada*.

Privilegirani način rada

U privilegiranom načinu rada procesoru su na raspolaganju sve instrukcije i sva sredstva sustava. U tom se načinu rada mogu izravno zabraniti i dozvoliti prekid (zasebnim instrukcijama), mogu se postavljati upravljački registri (`idtr`, `gdtr`, `ldtr`, `cr0-cr4`, `...`), može se mijenjati stanje procesora. Sustav Benu do inkrementa `Chapter_08_Processes` je koristio samo prsten 0 (privilegirani način rada).

Procesori koji se koriste u ugrađenim sustavima su vrlo često vrlo jednostavni procesori koji i

nemaju više različitih stanja, već sve rade u jednome. Stoga je pristup korišten u prijašnjim inkrementima prilagođen sustavima s takvim procesorima (takozvanim *mikrokontrolerima*).

Privilegirani način rada (u sustavima koji ga omogućuju) je pogodan za jezgrene funkcije koje moraju upravljati sustavom, podešavati podsustave i sklopovlje. Ponekad se i neki programi ili zbog učinkovitosti ili zbog važnosti također mogu izvoditi u ovome načinu rada procesora.

Neprivilegirani način rada

U neprivilegiranom načinu rada određene instrukcije procesora su nedostupne. Primjerice, dretva ne može zabraniti prekidanje, ne može pristupiti upravljačkim registrima, ne može pristupiti određenim napravama i spremničkim lokacijama. Ako se pojavi potreba za obavljanjem takve operacije, onda treba promijeniti način rada, tj. preći u jezgren način rada i zadanu operaciju obaviti jezgrenom funkcijom.

Mehanizam kojim dretva iz korisničkog načina rada poziva funkciju jezgre koju treba izvoditi u jezgrenom načinu rada, jest mehanizam *programski izazvanog prekida* (engl. *software interrupt*). Programskim prekidom ulazi se u privilegirani način rada te pokreće jedna od unaprijed pripremljenih i detaljnije provjerenih (ispitanih) jezgrenih funkcija. Dretva ne može "podmetnuti" svoj kôd za izvođenje u privilegiranom načinu rada. Može se pozvati samo ono što već postoji u jezgri. Na taj se način i u nekom složenijem višekorisničkom sustavu ostvaruje zaštita od zlonamjernih napada na sustav[1].

Programski prekid

Prelazak iz neprivilegiranog načina rada u privilegirani način rada tj. prelazak iz korisničkog načina rada u jezgren način rada, obavlja se mehanizmom prekida. Prekide mogu izazvati vanjske jedinice (kada se iz obrade prekida poziva odgovarajuću upravljački program), ali i sam procesor. Procesor izaziva prekide zbog grešaka u svom radu (primjerice dijeljenja s nulom, dohvat s nepostojeće adrese), ali i zbog programskih zahtjeva za prekidom – izazivanjem *programskog prekida*.

U skupu instrukcija (gotovo svakog) procesora postoji instrukcija za izazivanje programskog prekida. Kod arhitekture x86 postoji instrukcija `int n` kojom se izaziva programski prekid s prekidnim brojem `n`. Prihvat programskog prekida jednak je prihvatu sklopovski izazvanih prekida. Programski prekid, s obzirom na to da je izazvan u procesoru, se ne može maskirati, tj. ne može se zabraniti njegovo prihvaćanje i obradu (kao ni ostali prekidi izazvani u samom procesoru).

Prelazak iz privilegiranog načina rada u neprivilegirani (izlazak iz jezgre) obavlja se na isti način kao i povratak iz prekida, instrukcijom `iret`.

Prihvat prekida za x86 se izvodi različito ako se procesor u trenutku prekida već nalazio u privilegiranom načinu rada ili nije. Ako je u trenutku prekida procesor već bio u privilegiranom načinu rada, on ostaje u tom načinu rada i *minimalni kontekst* koji se sprema u postupku prihvata prekida (automatsko ponašanje procesora) se sprema na stog trenutne dretve (i dalje se koristi isti stog). U protivnom, ako je u trenutku prije prihvata prekida procesor bio u neprivilegiranom načinu rada, prihvatom prekida i prelaskom u privilegirani način rada aktivira se *prekidni stog* i na njega se sprema minimalni kontekst.

Zaštita na opisani način ima svoju cijenu. Osim dodatnog sklopovlja koje će omogućiti prijelaz iz jednog način rada u drugi te zaštitu spremničkog prostora, pri tim operacijama su potrebni i dodatni poslovi.

Osim očitih operacija (pohrana konteksta, obrada prekida, obnova konteksta) u nekim je arhi-

[1] Naravno, s obzirom na složenost današnjih sustava ni jezgra nije bez grešaka te se uvijek otkrivaju novi propusti u jezgri koji se mogu iskoristiti od strane napadača na sustav. Zato je vrlo bitno da se sustavi redovito ažuriraju zakrpama proizvođača operacijskih sustava kako bi rizik od uspješnog napada sveli na minimum.

tekturama (primjerice novijim Intelovim) proces zamjene načina rada popraćen i nekim unutarnjim (skrivenim) operacijama čija trajanja mogu biti i značajna (za tu operaciju). Problem nastaje u novijim (naprednim) arhitekturama koje koriste napredne tehnike upravljanja priručnim spremnicima za koje prekid predstavlja značajan poremećaj u radu. Prema nekim izvorima [osdev.org/CS], procesor *Pentium 4* potroši oko 2000 ciklusa samo za kućanske poslove (prihvat prekida te povratak u prekinutu dretvu nakon obrade). Iako sve skupa traje manje od mikrosekunde na procesorima čiji je radni takt dva ili više GHz, u okolnostima čestih prekida navedeno može značajno utjecati na učinkovitost sustava. Za usporedbu, Pentium II (200 MHz) je za istu operaciju potrošio oko 200 ciklusa, ali obzirom na frekvenciju na kojoj radi, operacija ipak dulje traje.

12.2. Pozivi jezgri preko programskih prekida

U kôdu u fazi `01_Syscall` uvodi se mehanizam programskog prekida za poziv jezgrenih funkcija, ali i dalje ostajući u privilegiranom načinu rada i za dretve, tj. pri prekidu se ne mijenja način rada te se kontekst dretve još uvijek sprema na njen stog. Međutim, po spremanju konteksta aktivira se zaseban stog koji se koristi u jezgrenim funkcijama. S obzirom na to da se sada jezgrene funkcije isključivo pozivaju mehanizmom prekida i da je stanje dretve pohranjeno u postupku prihvata prekida, svaka jezgrena funkcija se obavlja do kraja prije povratka u dretvu. Više nije potrebno pohranjivati i prekidni kontekst, sve jezgrene funkcije koriste isti stog – svaki put se prepisuje iznova – više ne vrijedi slika 11.5.

Slika 12.1. prikazuje stanje na stogovima (dretvinom i jezgrenom) na primjeru poziva sinkronizacijske funkcije `sem_post` mehanizmom programskog prekida.

Iz korisničke dretve poziva se jednostavnije sučelje `sem_post` za rad sa semaforima ostvareno u `api/pthread.c`. U toj funkciji se najprije na stog dretve spremaju parametri za jezgrenu funkciju te se izaziva programski prekid. U obradi prekida poziva se jezgrena funkcija za obradu programskih prekida. Ta funkcija prvo utvrđuje koju jezgrenu funkciju (operaciju) treba pozvati, dohvaća adresu vrha dretvina stoga na kojem su parametri te poziva zadanu jezgrenu funkciju s tom adresom kao jedinim parametrom.

Korištenjem programskog prekida za poziv jezgre započelo je pravo odvajanje dretvi od jezgre. Iako u ovoj fazi (i nekoliko idućih) i dalje sve dretve mogu pristupiti svim spremničkim lokacijama, ovakav poziv jezgre je temelj razdvajanja jezgre i korisničkih dretvi (programa).

Jezgrene funkcije pozivaju se preko potprograma `syscall` ostvarenog u `arch/i386/syscall.S`.

Isječak kôda 12.1. Chapter_08_Processes/01_Syscall/arch/i386/syscall.S

```
16 syscall:
17         int     $SOFT_IRQ
18         ret
```

S obzirom na to da je potprogram ostvaren u asembleru ne koristi se uobičajeni dodatni okvir stoga koji se inače stvara prevođenjem C programa. Drugim riječima, poziv tog potprograma iz C-a prema:

Isječak kôda 12.2. Chapter_08_Processes/01_Syscall/api/pthread.c

```
195         return syscall(SEM_POST, sem);
```

će generirati asembler oblika:

```
1         push    sem
2         push    $SEM_POST
3         call    syscall
4         add     $8, %esp        ;makni parametre sa stoga
```

Slika 12.1. Izgled stoga pri pozivu jezgrene funkcije `sem_post`

Na stogu će, trenutak prije izazivanja programskog prekida, biti redom od vrha stoga prema dnu:

1. adresa instrukcije iza `call`
2. vrijednost konstante `SEM_POST`
3. kazaljka na objekt semafora `sem`.

Imajući u vidu izgled prekidnog stoga, do adrese na kojoj se nalaze parametari koji su izvorno slani u jezgrenu funkciju dolazi se zbrajanjem trenutne adrese vrha stoga s veličinom elemenata koji su pohranjeni povrh parametara (što se i vidi na slici 12.1.). Ta se adresa izračunava te šalje kao jedini parametar u jezgrenu funkciju. Primjerice, za poziv `sys__sem_post` do jedinog parametra se dolazi preko adrese stoga gdje je on smješten pri pozivu jezgrene funkcije.

Isječak kôda 12.3. Chapter_08_Processes/01_Syscall/krenel/pthread.c

```
734
735         sem = *((sem_t **) p);
736
737         ASSERT_ERRNO_AND_EXIT(sem, EINVAL);
738
739         kthread = kthread_get_active();
740
```

```
741         kobj = sem->ptr;
742         ASSERT_ERRNO_AND_EXIT(kobj, EINVAL);
```

Kada jezgrena funkcija ima više parametara, kao pri slanju poruka:

Isječak kôda 12.4. Chapter_08_Processes/01_Syscall/api/pthread.c

```
12 int pthread_create(pthread_t *thread, pthread_attr_t *attr,
13                     void *(*start_routine)(void *), void *arg)
14 {
15         ASSERT_ERRNO_AND_RETURN(start_routine, EINVAL);
16
17         return syscall(PTHREAD_CREATE, thread, attr, start_routine, arg);
18 }
```

tada se do njih dolazi na isti način, s obzirom na to da su jedan do drugoga na vrhu stoga dretve:

Isječak kôda 12.5. Chapter_08_Processes/01_Syscall/krenel/pthread.c

```
27 int sys__pthread_create(void *p)
28 {
29         pthread_t *thread;
30         pthread_attr_t *attr;
31         void *(*start_routine)(void *);
32         void *arg;
```

Isječak kôda 12.6. Chapter_08_Processes/01_Syscall/krenel/pthread.c

```
41         thread = *((pthread_t **) p);           p += sizeof(pthread_t *);
42         attr = *((pthread_attr_t **) p);        p += sizeof(pthread_attr_t *);
43         start_routine = *((void **) p); p += sizeof(void *);
44         arg = *((void **) p);
```

Svi dosadašnji izravni pozivi jezgri oblika:

```
    sys__ ime_jezgrene_funkcije ( parametri );
```

zamijenjeni su drugim oblikom:

```
    syscall ( IME_JEZGRENE_FUNKCIJE, parametri );
```

te su sukladno svi `sys__*` pozivi preuređeni na gore prikazani način radi prihvata parametara.

Nadalje, svi pozivi prema jezgri (iz sloja *api*) su prilagođeni da koriste funkciju `syscall`.

Značajnije promjene napravljenje su u sloju *arch* te ponešto u `kernel/thread.c`. S obzirom na to da se u jezgru ulazi prekidom, a izlazi povratkom iz obrade prekida, kontekst dretve je prilagođen prekidnom kontekstu koji je opisan u strukturi `arch_context_t` koja je vidljiva i na slici 12.1.

Promjena konteksta s jedne dretve na drugu više se ne obavlja izravno funkcijom `arch_switch_to_thread` (koja se još jedino koristi u primjeru dretvi upravljanih izvan jezgre). Sada je dovoljno prije povratka iz jezgre (obrade prekida) definirati od kuda će se kontekst obnoviti, tj. s čijeg stoga (čije dretve). Navedeno se obavlja funkcijom `arch_select_thread`.

Obrada prekida, tj. jezgrena funkcija, se izvodi u kontekstu jezgrene dretve s vlastitim stogom (`k_stack` definiranim u `arch/i386/memory.c`) koji se aktivira pri pojavi prekida. Kako nema potrebe za čuvanjem tog konteksta po izlasku iz jezgre on se i ne pamti (iako se prije izlaska iz jezgre sve sa stoga i implicitno miče povratkom iz funkcija pa se zapravo i nema što sačuvati).

Promjena stoga s dretvenog na jezgren obavlja se u početku obrade prekida u dijelu kôda iza `arch_interrupts_common_routine` (u `arch/i386/interrupt.S`), dok se povratak

korištenju dretvina stoga ostvaruje nakon obrade prekida (nakon povratka iz `arch_interrupt_handler`).

12.3. Korisnički način rada

Poziv jezgrenih funkcija mehanizmom prekida započet je postupak odvajanja korisničkih dretvi i jezgrenih funkcija. Idući korak u odvajanju jest postavljanje procesora u *korisnički način rada* pri izvođenju kôda dretvi. Ta se promjena ostvaruje podešavanjem početnog registra stanja za nove dretve. Međutim, na arhitekturi x86 to ima značajnog utjecaja u načinu prihvata prekida i povratku iz prekida (tj. jezgre) kao i rada sa segmentima spremnika, odnosno njihovim opisnicima.

Arhitektura x86 ima sklopovsku potporu za višedretvenost u obliku opisnika dretvi (izvorno *task state segment – TSS*), dodatnih struktura podataka i operacija nad njima. Ipak, zbog ograničenja mehanizama povezanih uz *TSS*, uobičajeno je da se za sve dretve koristi isti *TSS* koji se ažurira svaki put kada se aktivna dretva mijenja. Iako *TSS* ima i predviđen prostor za spremanje konteksta dretve, od TSS-a se trenutno koristi jedino mogućnost definiranja adrese za spremanje konteksta prekinute dretve pri prihvatu prekida, odnosno pri prelasku u prsten 0 (u *privilegirani način rada*).

Promjene koje uzrokuje promjena načina rada dretvi (u prikazanom sustavu) utječu na:

- rad s opisnicima segmenata
- korištenje opisnika *TSS*
- prošireni (minimalni) kontekst dretvi koji se sprema pri prekidima
- promjena mjesta za spremanje konteksta dretve (sada se sprema u opisnik dretve, ne na njen stog).

U glavnoj tablici opisnika (*GDT*) potrebno je dodati dva nova opisnika koji se mogu koristiti iz korisničkog načina rada: jedan za instrukcije (`GDT_T_CODE`) i drugi za podatke dretvi (`GDT_T_DATA`). Opisnici segmenata za korisničke dretve (odnosno registri koji pokazuju na njih: `cs`, `ds`, `ss` i ostali) postaju sastavni dio konteksta dretvi koji se spremaju pri ulasku u jezgrenu funkciju te obnavljaju pri izlasku iz nje. U jezgrenim se funkcijama koriste drugi opisnici (za pristup iz privilegiranog načina rada). Rad s opisnicima segmenata opisan je u odjeljku 12.5.1.

U tablici opisnika prekida (*IDT*) potrebno je omogućiti izazivanje programskih prekida i iz korisničkog načina rada.

Korištenje opisnika *TSS* je stoga svedeno na definiranje adrese spremničkog prostora gdje će se spremiti kontekst dretve. Prije povratka iz jezgrene funkcije, u varijablu `arch_thr_context` se postavlja adresa s pohranjenim kontekstom dretve (koji će se učitati pri povratku u dretvu), a u TSS se elementi `.ss0` i `.esp0` (opisnik "stoga" razine 0) ažuriraju adresom iza mjesta za kontekst te dretve (kamo će se spremniti kontekst kada se ta dretva prekine prekidom), s obzirom na to da se pohranjivanje obavlja korištenjem stoga. Navedeno postavljanje se obavlja funkcijom `arch_select_thread` u `arch/i386/context.c`.

Isječak kôda 12.7. Chapter_08_Processes/02_User_mode/arch/i386/context.c

```
90 void arch_select_thread(context_t *context)
91 {
92         arch_thr_context = (void *) context;
93         arch_tss_update(((void *) &context->context) + sizeof(arch_context_t));
94 }
```

Kontekst dretve proširen je dodavanjem mjesta za registre koji pokazuju na opisnike segmenata,

ali i s dva dodatna elementa koji opisuju stog dretve u korisničkom načinu rada (neprivilegiranome). Naime, pri prekidu se automatski iz strukture *TSS* uzimaju `.ss0` i `.esp0` i postavljaju u registre `ss` i `esp` te se na taj stog sprema kontekst dretve. Prijašnje vrijednosti registara `ss` i `esp` (zatečene u radu dretve) se zato prve spremaju na taj stog (opisnik segmenata instrukcija, tj. registar `cs` se također kao i prije sprema na stog uz `eflags` i `eip`).

Kontekst koji se automatski pohranjuje (minimalni kontekst) sastoji se od registara redom: `ss`, `esp`, `eflags`, `cs`, `eip` + kôd greške za neke prekide. Dodatno, instrukcijama u `arch/i386/interrupt.S` se povrh toga spremaju registri opće namjene (`eax-edi`) te svi registri segmenata (`ds`, `es`, `fs`, `gs`), osim već automatski spremljenih (`ss`, `cs`).

Kontekst se više ne sprema na stog prekinute dretve već je postavljeno da se kontekst sprema izravno u opisnik dretve u element `.state.context` (tipa `context_t`). Razlog tome jest što se mjesto spremanja (adresa) konteksta treba postaviti u TSS prije povratka u dretvu. Dretva u tijeku svog rada (nakon) može mijenjati svoj stog te vrh stoga ne mora biti ono što je zatečeno prije povratka u dretvu (u pravilu i nije). Zato za svaku dretvu treba zauzeti posebno mjesto za njen kontekst – opisnik je jedan od logičkih odabira.

12.4. Odvajanje jezgre i programa u dvije cjeline

Idući korak u odvajanju jezgre i programa je odvajanje pri prevođenju, kako je prikazano u fazi `03_Programs_as_module`. Osim što se jezgra i programi zasebno prevode kao i u prijašnjim koracima, sada se od jezgre stvara jedna cjelina (izlazna datoteka `kernel.elf`), a od programa druga (`prog.bin`). Te se dvije cjeline više ne spajaju u jednu datoteku već se sustav sastoji od dvije (koje se predaju pokretaču ili emulatoru).

Korištenjem mogućnosti QEMU-a jezgra se, kao i prije, učitava na zadanu adresu (zadanu u slici sustava) i pokreće. Programi se sada učitavaju kao modul na adresu koju određuje QEMU. Zato se pri inicijalizaciji jezgre (pri pokretanju) programi prvo kopiraju na adresu za koju su priređeni, a tek onda pokreću.

Pri povezivanju programa u `prog.bin` koristi se nova datoteka `user.ld`.

Isječak kôda 12.8. Chapter_08_Processes/03_Programs_as_module/arch/i386/boot/user.ld

```
3  /* Its parsed as C before used in linking! */
4
5  OUTPUT_FORMAT("binary")
6
7  ENTRY(prog_init)
8
9  SECTIONS {
10
11         .user PROG_START_ADDR: /* prepare for absolute address */
12         {
13                 user_code = .;
14
15                 /* header */
16                 * ( *.program_header* )
17
18                 /* instructions */
19                 * (.text*)
20
21                 user_data = .;
22
23                 /* read only data (constants), initialized global variables */
24                 * ( .rodata* .data* )
25
26                 user_bss = .;
27
28                 /* uninitialized global variables (or initialized with 0) */
```

```
29                 * ( .bss* COMMON* )
30
31                 . = ALIGN (4096);
32                 user_end = .;
33         }
34
35         #ifndef DEBUG
36                 /DISCARD/ : { *(*) }
37         #endif
38         /DISCARD/ : { *(.comment*) } /* gcc info is discarded */
39         /DISCARD/ : { *(.eh_frame*) } /* not used */
40         /DISCARD/ : { *(.note*) } /* not used */
```

Modul s programima je pripremljen u binarnom obliku (QEMU ga učitava u memoriju bez interpretacije i promjene). Programi se pripremaju za adresu definiranu konstantom `PROG_START_ADDR` (vrijednost 0x200000 definirana u datoteci s postavkama `config.ini`). Na početku bloka s programima nalazi se struktura s informacijama o programima (podatkovni dio datoteke `prog_info.o`):

Isječak kôda 12.9. Chapter_08_Processes/03_Programs_as_module/api/prog_info.c

```
8  /* symbols from user.ld */
9  extern char user_code, user_end, user_heap, user_stack;
10
11 extern int PROG_START_FUNC(char *args[]);
12
13 prog_info_t pi __attribute__((section(".program_header"))) =
14 {
15         .magic =        { PMAGIC1, ~PMAGIC1, PMAGIC2, ~PMAGIC2 },
16         .type =         MS_PROGRAM,
17         .start =        &user_code, /* from user.ld */
18         .end =          &user_end,  /* from user.ld */
19         .name =         PROG_START_FUNC_NAME,
20         .init =         prog_init,
21         .entry =        PROG_START_FUNC,
22         .param =        NULL,
23         .exit =         pthread_exit,
24         .prio =         THR_DEFAULT_PRIO,
25 };
```

Pronalazak gdje je modul učitan napravljen je na najednostavniji način: pretragom spremnika za identifikatorom modula (koji se nalazi na početku samog modula). Nakon pronalaska modul se kopira na adresu 0x200000 za koju su program i pripremljeni (u funkciji `k_memory_init` datoteke `kernel/memory.c`). S obzirom na to da se na početku bloka s programima nalazi navedena struktura, ona se koristi pri inicijalizaciji okoline za programe (dinamičko upravljanje spremnikom u programima) kao i prije u `kthreads_init`.

Ovakav način učitavanja i pokretanja programa podsjeća na "prave" operacijske sustave kod kojih se program učitava s nekog medija (primjerice diska) u radni spremnik u kojem je priređena okolina za njegovo izvođenje. Umjesto nekog medija, ovdje se koristi spremnik, odnosno dio spremnika u koji je QEMU učitao modul s programima.

12.5. Programi kao zaseban proces

Idući korak u odvajanju jezgre i programa odvaja *adresni prostor* jezgre od programa (faza `04_Programs_as_process`). Korištenjem sklopovske potpore u obliku *rada sa segmentima* i njihovim *opisnicima*, adresni prostor programa je ograničen na zadane segmente. Time su programi "fizički" odvojeni od jezgre, odnosno nalaze se u posebnom procesu. Za takvo razdvajanje potrebno je bolje upoznati osnovne načine rada sa segmentima u arhitekturi x86.

12.5.1. Spremnički segmenti arhitekture x86

Pri svakom dohvatu podataka iz spremnika i pri svakom pohranjivanju podataka u spremnik u arhitekturi x86 koriste se *opisnici segmenata*. Pri dohvatu instrukcija koristi se *opisnik segmenata instrukcija*, pri radu s podacima u spremniku koristi se *opisnik segmenta podataka*, pri korištenju stoga koristi se *opisnik segmenta stoga*. Svi opisnici trebaju biti definirani u tablicama *GDT* ili *LDT* (engl. *local descriptor table*). Opisnici u GDT-u koji se koriste u Benu prikazani su u tablici 12.1.

Tablica 12.1. Tablica opisnika segmenata (GDT, LDT)

indeks	namjena	razina	početak	veličina	koriste ga registri
0					
1	jezgra: kod	0	0	0xFFFFFFFF	cs
2	jezgra: podaci	0	0	0xFFFFFFFF	ds, ss, es, fs, gs
3	proces: kod	3	proc.start	proc.size − 1	cs
4	proces: podaci	3	proc.start	proc.size − 1	ds, ss, es, fs, gs
5	jezgra: TSS	0	&tss	sizeof(tss_t) − 1	tr

Procesor koristi nekoliko dodatnih registara za podršku rada sa segmentima i upravljanje spremnikom na taj način, prikazani u tablici 12.2.

Tablica 12.2. Registri x386 procesora koji se koriste u upravljanju spremnikom

registar	namjena
eflags	određuje način rada procesora (zastavice IOPL)
gdtr	pokazuje na tablicu GDT
ldtr	pokazuje na tablicu LDT
cs	pokazuje na redak u GDT ili LDT, koristi se pri dohvatu instrukcija
ds	pokazuje na redak u GDT ili LDT, koristi se pri korištenju podataka iz spremnika (npr. instrukcija `movl $42, (%eax)`)
ss	pokazuje na redak u GDT ili LDT, koristi se pri radu sa stogom (npr. instrukcija `pushal`)
es, fs, gs	pokazuje na redak u GDT ili LDT, koristi se pri radu s podacima (npr. instrukcija `movl $42, %fs:(%eax)`)
tr	pokazuje na redak u GDT ili LDT, koristi se pri prihvatu prekida za spremanje konteksta prekinute dretve

Svaki od registara (`cs`, `ss`, `...`) pokazuje na opisnik segmenta tako da identificira tablicu u kojoj se opisnik nalazi (GDT ili LDT), sadrži indeks opisnika u toj tablici (redni broj opisnika) te definira potrebnu razinu privilegija za korištenje tog opisnika.

U kôdu je korišten samo *GDT*. *LDT* je predviđen za korištenje po pojedinom procesu. Umjesto tablice *LDT* korištena je tablica *GDT* čak i u kasnijim fazama kada u sustavu ima više procesa – tada se pri povratku u dretvu svaki put ažurira *GDT*.

Svaki opisnik (u tablicama *GDT* i *LDT*) sastoji se od:

- fizičke adrese početka segmenta
- veličine segmenta
- prava pristupa (za čitanje/pisanje/izvođenje)
- potrebne razine privilegija za pristup segmentu.

Primjeri načina korištenja opisnika prikazan je na instrukcijama za dohvat podatka:

```
movl    adresa_varijable, %eax
movl    (%ebx), %eax
movl    %eax, 8(%ebx)
```

Pri obavljanju ove operacije (nakon što je dohvaćena instrukcija korištenjem sličnih pravila) najprije se izračunava prava *fizička adresa* (još se naziva i *apsolutna adresa*) korištenjem zadane *logičke adrese* (*relativne adrese*, *pomaka*, engl. *offset*). Fizička adresa se izračunava kao suma adrese početka segmenta (koji se uzima iz opisnika na koji pokazuje `ds` registar za podatke) i zadane adrese (vrijednost `adresa_varijable`), tj.

fizička_adresa (adresa_varijable) = početna_adresa_segmenta + adresa_varijable.

Potom se provjerava spada li izračunata fizička adresa zaista u segment, tj. izlazi li možda iz njega. Provjera se obavlja korištenjem opisnika segmenta, tj. početne adrese i veličine segmenta. Ako je adresa valjana (pripada segmentu) onda se nastavlja s idućom provjerom prava pristupa (tražena razina privilegija tekuće dretve). Ako je i iduća provjera uspješna operacija dohvata se provodi. U protivnom, ako adresa ne pripada segmentu već je izvan njega ili trenutna dretva nema potrebnu razinu privilegija, procesor izaziva prekid zbog narušene sigurnosti sustava (engl. *general protection fault*).

Svaki od opisnika segmenata može pokazivati na drugi segment spremnika. Međutim, Benu koristi navedene mehanizme samo radi izolacije procesa od jezgre te jednog procesa od drugoga. Zato svi opisnici segmenata za isti proces pokazuju na isti segment (početak i veličinu). Zapravo su dovoljna samo dva opisnika u tablici *GDT*: jedan za instrukcije (`cs`) i jedan za podatke (`ds=ss=es=fs=gs`). Razlika u ta dva opisnika segmenta jest jedino u pravima, s obzirom na to da segment instrukcija mora imati oznaku da se radi o segmentu instrukcija.

Slika 12.2. prikazuje odnos fizičke adrese (koja se stavlja na sabirnicu) i logičke adrese koju stvara program u svom izvođenju koji interno radi s logičkim adresama. Korištenjem segmentacije procesu se dozvoljava pristup adresnom prostoru samo unutar zadanog segmenta, na slici prikazanog sivom bojom. Ostali dijelovi spremnika su mu nedostupni.

Slika 12.3. prikazuje odnos fizičkih i logičkih adresa na primjeru programa koji se nalazi na disku nekog sustava, procesa koji nastaje pokretanjem tog programa korištenjem fizičkog adresiranja (bez segmentacije) te procesa koji nastaje pokretanjem tog programa koji koristi logičke adrese (sa segmentacijom ili straničenjem). Instrukcije korištene u primjeru su općenitije (ne pripadaju arhitekturi x86, ali je načelo isto).

12.5.2. Ostvarenje procesa

Do sada (prijašnji koraci) su također koristili opisnike segmenata (oni se uvijek koriste), ali s početnom adresom postavljenom na nulu te veličinom segmenta koja prekriva cijeli spremnički prostor (4 GB). Ovakvim postavkama opisnika segmenata (0–4 GB) je zaštita preko segmenata zapravo isključena – sve su logičke adrese ujedno bile i fizičke i svaka je provjera bila uspješna (svaka 32-bitovna adresa spada u navedeni segment). U fazi `04_Programs_as_process` napravljeno je odvajanje programa (procesa) od jezgre prikazanim mehanizmima upravljanja spremnikom korištenjem segmentacije. Logički adresni prostor procesa započinje adresom 0 i završava dodijeljenom mu veličinom (L sa slike 12.2.).

Prilagođena skripta za stvaranje programa `user.ld` (koji će postati proces s logičkim adre-

Slika 12.2. Odnos fizičke i logičke adrese

program na disku (prije pokretanja)

```
  0  (početak)
     .
 20  LDR R0, (100)
 24  LDR R1, (104)
 28  ADD R2, R0, R1
 32  STR R2, (120)
 34  B 80
     .
     .
 80  CMP R0, R3
     .
     .
100  DD 5
104  DD 7
     .
120  DD 0
```

proces (fizičke adrese) učitan na PA=1000

```
1000  (početak)
      .
1020  LDR R0, (1100)
1024  LDR R1, (1104)
1028  ADD R2, R0, R1
1032  STR R2, (1120)
1034  B 1080
      .
      .
1080  CMP R0, R3
      .
      .
1100  DD 5
1104  DD 7
      .
1120  DD 0
      .
      .
1500  (vrh stoga)
```

proces (logičke adrese)

```
  0  (početak)
     .
 20  LDR R0, (100)
 24  LDR R1, (104)
 28  ADD R2, R0, R1
 32  STR R2, (120)
 34  B 80
     .
     .
 80  CMP R0, R3
     .
     .
100  DD 5
104  DD 7
     .
120  DD 0
     .
     .
500  (vrh stoga)
```

Slika 12.3. Odnos fizičke i logičke adrese na primjeru

sama) sada definira te statički zauzme sav potreban adresni prostor za proces.

Isječak kôda 12.10. Chapter_08_Processes/04_Programs_as_process/arch/i386/boot/user.ld

```
/* Its parsed as C before used in linking! */

OUTPUT_FORMAT("binary")

ENTRY(prog_init)

SECTIONS {

        /* preallocate all space that is required at runtime! */
        .user 0:
        {
                user_code = .; /* == 0 */

                /* program/module header */
                * ( *.program_header* )

                /* instructions */
                * (.text*)

                user_data = .;

                /* read only data (constants), initialized global variables */
                * ( .rodata* .data* )

                user_bss = .;

                /* uninitialized global variables (or initialized with 0) */
                * ( .bss* COMMON* )

                . = ALIGN (4096);

                user_heap = .;
                . += 0x10000; /* allocate space for heap */

                user_stack = .;
                . += 0x10000; /* allocate space for stack */
                user_end = .;
        }

        #ifndef DEBUG
                /DISCARD/ : { *(*) }
        #endif
        /DISCARD/ : { *(.comment*) } /* gcc info is discarded */
        /DISCARD/ : { *(.eh_frame*) } /* not used */
        /DISCARD/ : { *(.note*) } /* not used */
```

Teoretski bi se prostor za *gomilu* i stog (adrese `user_heap`, `user_stack`) mogao izbaciti iz tog opisa i dinamički pridodati na kraj segmenta. Međutim, s obzirom na to da se programi učitavaju kao *modul* (operacija koju provodi QEMU) nije poznato što se nalazi neposredno iza učitanog modula (može biti još nešto). Zato se unaprijed zauzme sav potrebni prostor.

Drugi način (jednako dobar ili bolji) jest da se (kao u prethodnoj fazi) najprije kopira cijeli učitani modul na drugu adresu i tamo proširi potrebnim prostorom za gomilu i stog (što je i napravljeno u zadnjoj fazi).

Promjene u kôdu jezgre zbog različitih načina adresiranja u jezgri i programima se prvenstveno odnose na rad s parametrima pri pozivu jezgrenih funkcija. Naime, sada je potrebno voditi računa o tome o kakvoj se adresi radi – fizičkoj ili logičkoj. Pretvorba adresa obavlja se zbrajanjem ili oduzimanjem zadane adrese od adrese početka procesa. Idući primjer pretvorbe adrese za jednu jezgrenu funkciju karakterističan je i za ostale pozive jezgri.

Isječak kôda 12.11. Chapter_08_Processes/04_Programs_as_process/kernel/pthread.c

```
165 int sys__pthread_self(void *p)
166 {
167         pthread_t *thread;
168
169         thread = U2K_GET_ADR(*((void **) p), kthread_get_process(NULL));
170
171         ASSERT_ERRNO_AND_EXIT(thread, ESRCH);
172
173         thread->ptr = kthread_get_active();
174         thread->id = kthread_get_id(NULL);
175
176         EXIT(EXIT_SUCCESS);
177 }
```

Značajnije promjene napravljene su i u podsustavu za dretve, s obzirom na to da sada treba paziti da sve potrebno bude dretvi dostupno u njenim adresama. Primjerice, pri stvaranju početnog konteksta za novu dretvu, primljeni parametri (adresa početne funkcije, adresa stoga, parametar, ...) moraju ostati u logičkom obliku, dok za jezgru treba napraviti pretvorbu (da se mogu staviti parametri na stog dretve).

Nadalje, dodan je opisnik procesa koji sadrži potrebne parametre za upravljanje.

Isječak kôda 12.12. Chapter_08_Processes/04_Programs_as_process/kernel/memory.h

```
56 struct _kprocess_t_
57 {
58         mseg_t        m;
59                       /* memory segment this process occupies */
60
61         kprog_t      *prog;
62                       /* link to associated program */
63
64         process_t    *proc;
65                       /* process header - at start of process memory */
66
67         char          name[16]; /* program name */
68
69         void         *heap; /* physical address of heap area */
70         size_t        heap_size;
71
72         void         *stack; /* physical address of stack area */
73         size_t        stack_size;
74         size_t        thread_stack_size;
75         uint         *smap; /* bitmap for stack allocation */
76                       /* allocation unit = thread_stack */
77                       /* allocation units = stack_size / thread_stack */
78         uint          smap_size;
79
80         uint          prio;     /* default priority for threads */
81
82         int           thread_count;
83
84         list_t        kobjects;
85                       /* kobject_t elements */
86 };
```

Osim informacija o adresnom prostoru procesa (`.m`, `.prog`, `.proc`), u opisniku se nalazi i brojač dretvi koje pripadaju procesu (`.thread_count`, da se proces može ugasiti sa zadnjom dretvom) te opisnik za upravljanje spremnikom za stogove dretvi (`.smap` i `.smap_size`) koji koristi bitmapu za oznaku praznih dijelova spremnika rezerviranog za stogove dretvi procesa (pogledati funkcije `kprocess_stack_alloc` i `kprocess_stack_free` u `kernel/memory.c`).

Opisnik za dinamičko upravljanje spremnikom za gomilu (za dretve procesa) nalazi se u `.proc` elementu (`.mpool`). On se izravno koristi iz dretve, tj. sve su adrese logičke, relativne u odnosu na početak procesa – zato je i inicijalizacija gomile napravljena iz konteksta procesa, kao prva operacija početne dretve procesa u `prog_init` funkciji u `api/prog_info.c`.

12.6. Statički procesi

Faza `05_Static_processes` dijeli svaki program iz `programs` direktorija u zasebni proces. Pristup je sličan kao i u prethodnoj fazi uz to da se svaki program zasebno prevodi kao zasebni modul, uključujući sav potreban spremnički prostor za proces. Zato pokretanje pojedinog procesa zahtijeva samo stvaranje potrebnih opisnika te pokretanja početne dretve. Značajnije promjene napravljene su u `config.ini` omogućujući definiranje više parametara za prevođenje svakog programa zasebno.

Procesi su kao i u prethodnoj fazi statički definirani prevođenjem i izvode se sa spremničkog prostora u koji ih je QEMU učitao. Svaki se program zato može pokrenuti samo jednom (samo jedna instanca istog programa).

12.7. Dinamički procesi

Proširenje prethodne faze mogućnošću dinamičkog pokretanja procesa kakav postoji u stvarnim operacijskim sustavima izvedeno je u fazi `06_Dynamic_processes` kopiranjem osnovnog dijela procesa (učitanog kao modul) na novu spremničku lokaciju za novi proces, gdje se osnovni dio proširuje prostorom za gomilu i stogove. Na ovaj se način može pokrenuti i više instanci istog programa istovremeno.

Nedostatak ovakvog dinamičkog upravljanja spremnikom jest u nemogućnosti naknadnog proširenja adresnog prostora procesa. Naime, pri stvaranju procesa se definira njegova veličina i kasnije ju nije moguće mijenjati.

Promjena za podršku "dinamičkih procesa" nije velika, dijelom u izvornim kodovima, a dijelom u skripti za povezivanje programa `user.ld` koja više ne zauzima sav potreban adresni prostor za proces i u slici sustava, već se on dodaje samo pri pokretanju sustava.

12.8. Straničenje

Iako *straničenje* nije ostvareno u Benu, ono je vrlo bitno te je stoga ipak detaljnije objašnjeno ovdje.

Upravljanje spremnikom dinamičkom metodom, kako je to prikazano u kôdu u fazi `06_Dynamic_processes` `06_Dynamic_processes`, rješava problem odvajanja programa (procesa) od jezgre te se greške pojedinih programa mogu izolirati. Ipak, navedeno rješenje ima i nekoliko nedostataka.

Nedostaci dinamičkog upravljanja spremnikom uključuju:

- nemogućnost naknadnog povećanja i smanjenja spremničkog prostora dodijeljenog pojedinom procesu
- problem *fragmentacije* (neiskorišteni dijelovi spremnika su razbacani i ne mogu se spojiti u jedan veći, potreban za pokretanje novih procesa)
- uvišestručavanje dijelova s instrukcijama pri višestrukim pokretanjima istog programa
- nemogućnost učinkovitog korištenja pomoćnog spremnika (primjerice diska) kada je glavni spremnik premali da obuhvati sve podatke svih procesa.

U nekim primjenama (primjerice u ugrađenim sustavima) navedeni nedostaci možda i nisu toliko izraženi te je dinamičko upravljanje spremnikom zadovoljavajuće. Za zahtjevnije sustave treba koristiti napredniji način upravljanja spremnikom – koristiti *straničenje* koje iziskuje složenu sklopovsku potporu.

Osnovna ideja straničenja jest da se logički spremnički prostor procesa podijeli u mnoštvo malih jednakih dijelova koji se nazivaju *stranice*, da se fizički spremnik podijeli u dijelove istih veličina koji se nazivaju *okviri* te da se korištenjem sklopovske potpore svaka logička adresa, koja se odnosi na određeni dio stranice procesa prevede u fizičku adresu (dio nekog okvira). Navedeno prikazuje slika 12.4.

Slika 12.4. Podjela na stranice i okvire

Veličina jedne stranice mora odgovarati veličini jednog okvira. Stranica i okviri moraju biti poravnati na adrese koje su višekratnici veličine stranice koje su potencija broja 2 (tj. veličina stranice je jednaka 2^p). Drugim riječima, adresa početka svake stranice (i okvira) ima nule u najnižih p bitova, dok viši bitovi predstavljaju redni broj (indeks) stranice i okvira.

Svaka logička adresa (primjerice adresa jedne varijable) može se podijeliti na dva dijela: gornjih r-bita koji definiraju redni broj stranice te donjih p-bita koji određuju odmak od početka stranice. Budući da su stranice i okviri jednakih veličina, pri pretvorbi iz logičke u fizičku adresu dovoljno je pretvoriti redni broj stranice (r-bita) u redni broj okvira (q-bita) u kojem se stranica nalazi. Ako se logička adresa u binarnom obliku zapiše s $LA = [rb_str : odmak]$ tada se njena fizička adresa može zapisati s $FA = [rb_okv : odmak]$, odnosno za pretvorbu adresa potrebno je odrediti redni broj okvira za zadani redni broj stranice: $rb_okv = f(rb_str)$, tj. $FA = [f(rb_str) : odmak]$.

Logička adresa i fizička adresa ne moraju biti jednake duljine. Na primjer, logička adresa se može iskazati sa 64 bita dok se fizička adresa može iskazati sa samo 32 bita. Ipak je uobičajeno da su adrese jednakih duljina (barem načelno podržano i sklopovljem).

Pretvorba rednog broja stranice u redni broj okvira obavlja se sklopovski, korištenjem *tablice prevođenja*. Tablicu prevođenja stvara i ažurira operacijski sustav za svaki novi proces, a koristi ju *sklop za prevođenje adresa*. Opća ideja pretvorbe logičkih adresa u fizičke korištenjem straničenja, tj. tablice prevođenja prikazana je na slici 12.5.

Slika 12.5. Pretvorba adresa preko tablice prevođenja

Pri pokretanju procesa operacijski sustav stvara tablicu prevođenja za taj proces te podešava odgovarajuće upravljačke registre procesora prije povratka u dretve procesa (podešava sklop za prevođenje adresa, tj. registar koji pokazuje na početak tablice prevođenja).

Tablica prevođenja ne mora opisivati sav mogući adresni prostor procesa jer bi tada sama tablica zauzimala znatan dio spremnika. Primjerice, ako su adrese 32-bitovne tada je mogući adresni prostor velik 4 GB. Ako je stranica velika 4 KB tada za opis 4GB treba 4 GB / 4 KB = 1 M stranica (2^{20}). Ako je opisnik u tablici prevođenja velik 32 bita tada bi za tablicu bilo potrebno 4 MB spremničkog prostora (32 bita $\times$ 2^{20} = 4 B $\times$ 2^{20} = 4 MB). Gotovo svi procesi su manji od 4 GB te bi ovakve potpune tablice bile suviše zahtjevne na sustav (previše spremničkog prostora). Tablice prevođenja se zbog toga ne popunjavaju za cijeli adresni prostor već samo za dio koji je zauzet (koristi se ili je zauzet). Da bi se olakšalo sklopovsko rješenje sažimanja tablice, tablica se izgrađuje hijerarhijski. Na slici 12.6. prikazan je jednostavan primjer s dvorazinskom tablicom prevođenja.

Redni broj stranice dijeli se na dva dijela: prvi definira indeks retka u početnoj (glavnoj) tablici prevođenja dok drugi definira indeks retka u odabranoj tablici prevođenja u drugoj razini. Odabrani redak u tablici druge razine sadrži pravi indeks okvira u kojem se stranica nalazi, a koji se koristi pri pretvorbi logičke u fizičku adresu.

Kada proces ne bi koristio sav adresni prostor, neke tablice u drugoj razini ne bi trebale ni postojati. Tada bi u odgovarajućim retcima početne tablice koji pokazuju na tablice u drugoj razini s posebnim bitovima (zastavicama) trebalo naznačiti da takva tablica ne postoji.

Uobičajeno je da svaka od tablica (početna te svaka u drugoj razini) i sama zauzima po jednu stranicu. Takvo ostvarenje će u svim tablicama imati retke istog formata. Također i početna

Slika 12.6. Primjer prevođenja korištenjem hijerarhijske tablice

adresa tablice prevođenja koja se učitava u neki registar procesora (njegova dijela koji se koristi za sklopovsko prevođenje adresa) se može proširiti istim zastavicama. Ovakvim pristupom se sve stranice pa i one koje su dio tablice prevođenja, dohvaćaju na isti način.

Svaki zapis u tablici prevođenja sadrži, pored adrese pripadajućeg okvira i dodatne bitove – zastavice. Jedna od tih zastavica jest *bit prisutnosti* (engl. *validity bit*) koji kaže nalazi li se uopće stranica na koju pokazuje redak tablice u radnom spremniku (u nekom okviru). Ako se stranica koju je zatražio neki proces ne nalazi u radnom spremniku sklop za prevođenje adresa mora izazvati prekid (prekid zbog promašaja) jer se adresa ne može prevesti. Operacijski sustav u obradi tog prekida može s pomoćnog spremnika dohvatiti tu stranicu, staviti ju u prazni okvir (po potrebi i isprazniti neki za tu potrebu) te ažurirati tablicu prevođenja tog procesa. Povratkom u prekinuti proces instrukcija koje je izazvala prekid se ponavlja.

Na navedeni način može se ostvariti takozvano *straničenje na zahtjev* (engl. *demand paging*) kod kojeg se pri pokretanju procesa učita samo nekoliko njegovih stranica. Sve ostale se učitavaju s pomoćnog spremnika tek po potrebi (u obrada prekida promašaja tog procesa).

Jezgra ipak mora dodatnom strukturom podataka pratiti koje je segmente logičkog adresnog prostora procesa dodijelila ili zauzela za proces. Kada se dogodi prekid i ustanovi se da je tražena adresa izvan svih takvih segmenata proces se prekida jer se očito dogodila greška – traži se adresa koja nije dodijeljena procesu (engl. *segmentation fault*).

Druge zastavice pobliže opisuju ciljanu stranicu. Primjerice, opisuju dozvole pristupa, označavaju je li stranica uopće korištena do sada, je li mijenjana u odnosu na kopiju na pomoćnom spremniku i slično.

Problemi dinamičkog upravljanja spremnikom, navedeni na početku ovog odjeljka, straničenjem se u potpunosti mogu riješiti. Straničenje omogućava učinkovito korištenje pomoćnog spremnika za pohranu stranica koje se trenutno ne koriste. Višestrukim pokretanjem istog programa ne moraju se nanovo učitavati isti segmenti instrukcija već se prethodno učitani mogu mapirati u tablicama prevođenja svih idućih procesa (koji koriste iste segmente instrukcija). Problem fragmentacije ne postoji jer je granulacija okvira jako mala i dodjeljivanje se obavlja

po okvirima.

Dinamičko dodavanje spremničkog prostora postojećem procesu je također ostvarivo. Primjerice, ako se instrukcije postave na početak logičkog adresnog prostora procesa, stog na kraju, sredina ostaje za podatke (gomilu). U početku je ta sredina gotovo prazna, ali se daljnjim radom procesa mogu zahtijevati novi segmenti za koje operacijski sustav stvara stranice i dodaje ih u tablicu prevođenja, tj. povećava spremnički prostor procesa.

12.8.1. O ostvarenju straničenja

Straničenje nije ostvareno u okviru sustava Benu.

Kada bi se krenulo u ostvarenje straničenja osim uobičajenih operacija za stvaranje i ažuriranje tablice prevođenja treba voditi brigu i o prekidima. Naime, prekidom se prelazi u nadgledni način rada. Ali što je sa straničenjem pri prekidu? U arhitekturi x86 sustav straničenja podešen za prekinuti proces ostaje aktivan i u trenutku prihvata prekida. Odluka je projektanta kako će postupiti dalje. On može ostati u istom sustavu straničenja prekinutog procesa ili aktivirati drugi, jezgren sustav straničenja (s vlastitom tablicom prevođenja) ili najprije isključiti straničenje pa tek onda započeti s izvođenjem jezgrenih funkcija. U prvom slučaju (uobičajenom za operacijske sustave) svaki proces na isti način mora mapirati spremnički prostor jezgre. Uobičajeno je da se početni dio logičkog adresnog prostora (ili krajnji dio) "dodijeli jezgri", a ostatak procesu. To znači da se "jezgra" može dohvatiti korištenjem tablice prevođenja svakog procesa. Da bi se procesima onemogućilo da sami pristupaju tom dijelu svog adresnog prostora te se stranice označuju kao stranice sustava te se svaki pokušaj pristupa iz korisničkog načina rada prekida izazivanjem prekida. Glavnina struktura podataka jezgre (ako ne i sva) i jezgrene funkcije trebaju biti u tom segmentu spremnika (opisanim dijelom tablica prevođenja). Primjerice, logički adresni prostor od 1. do 3. GB-a može pripadati procesu dok se zadnji GB zauzima za jezgru ili obratno (1. GB za jezgru, zadnja 3 za proces).

12.8.2. Nedostaci straničenja

Iako rješava probleme statičkog i dinamičkog upravljanja spremnikom i straničenje ima neke nedostatke u određenim okolinama.

Straničenje zahtijeva složenu sklopovsku potporu te se ne koristi kod jednostavnih procesora jer bi tada njihova cijena bila znatno veća. Za složenije procesore taj se dodatak gotovo i ne primjećuje (u ukupnoj veličini procesora, tj. broju tranzistora).

U sustavima koji pored glavnog, radnog spremnika, koriste i pomoćni spremnik, izvođenje procesa se zbog promašaja ponekad može znatno odgoditi. Dohvat stranice procesa s pomoćnog spremnika (diska) će potrajati bar desetak milisekundi. Iako će za to vrijeme procesor moći obavljati druge procese, navedena odgoda jednog procesa ograničava uporabu sustava, tj. one se ne može koristiti u sustavima koji zahtijevaju poštivanje vremenskih ograničenja (SRSV). Ipak, ako se kritični program u cjelini stalno drži u radnom spremniku (posebnim zahtjevima operacijskom sustavu), tada se taj program ipak može upotrijebiti i za kritične zadatke. Zadaća je arhitekta sustava da odredi koji dijelovi sustava (programi i dijelovi jezgre) moraju ostati cijelo vrijeme u radnom spremniku, a koji nisu kritični i kojima se može dopustiti poneka odgoda u izvođenju. Naravno, operacijski sustav mora podržavati mogućnosti zaključavanja nekih stranica ili cijelih procesa u radni spremnik. Primjerice, POSIX pozivi `mlock` i `mlockall` te *Win32* pozivi `VirtualLock` te `SetProcessWorkingSetSize` mogu pomoći za takve sustave (iako treba provjeriti jesu li za zadani sustav to samo preporuke operacijskom sustavu ili naredbe).

Pitanja za vježbu 12

1. Što su to dretve? Što je to jezgra? U čemu se razlikuje izvođenje (korisničke) dretve od izvođenja jezgrene funkcije?
2. Što su to procesi? Koje mehanizme pružaju?
3. Koji su načini ostvarenja podrške za procese?
4. Što je to logička adresa, a što fizička? Kada se neki program prevodi u strojni oblik, koje se adrese koriste? Zašto?
5. Korištenjem segmenata (sklopovske potpore u obliku zbrajala) adrese koje program generira pretvaraju se iz logičkih u fizičke. Kako? Koji su nedostaci ostvarenja procesa nad ovim mehanizmom?
6. Koji su osnovna načela straničenja? Kako se logička adresa pretvara u fizičku?
7. Kako se straničenjem ostvaruje upravljanje spremnikom?
8. Zašto se straničenje rijetko koristi u ugrađenim sustavima?
9. Koji su nedostaci straničenja? Na što treba paziti ako se straničenje koristi u SRSV-u?

13. Zasnivanje ugrađenih računalnih sustava

Ugrađeni računalni sustavi mogu se izgraditi na razne načine. Jedan od njih je izgradnja od nule. Međutim, kao što je to već rečeno, takav je način najdulji, a možda i najskuplji. Drugi načini uključuju korištenje dostupnih sustava. U ovom poglavlju razmatraju se osnovna svojstva nekih trenutno dostupnih sustava te ostali načini zasnivanja računalnih sustava za ugradbena računala.

13.1. Pregled nekih operacijskih sustava projektiranih za ugrađene sustave

U novije vrijeme pojavljuju se novi operacijski sustavi koji pokrivaju segment takozvanih prijenosnih i ručnih računala, bilo samostojećih ili povezanih s mogućnostima mobilnih telefona ili drugih uređaja. Primjeri takvih sustava su "pametni" mobilni telefoni (engl. *smartphones*), multimedijalni uređaji, čitači elektroničkih knjiga, ručna računala. Primjeri operacijskih sustava koje koriste navedeni uređaji su iOS, Android i Windows Phone 8.

Ne može se reći da su prethodni sustavi tipični ugrađeni sustavi koji su razmatrani do sada. Dapače, navedeni su po svojstvima bliži općim operacijskim sustavima za stolna računala. Međutim, i ti operacijski sustavi nastoje proširiti područja svoje primjene u ugrađenim sustavima, primjerice za uređaje u automobilima, za "pametne" televizore i slično. Zato se i razvijaju u smjeru postizanja bar minimalnih svojstava potrebnih za takve sustave, tj. nastoje ostvariti mogućnosti za primjenu u sustavima s blagim vremenskim ograničenjima.

U nastavku je najprije malo detaljnije opisan FreeRTOS te potom QNX Neutrino, VxWorks i ostali. Posebnost FreeRTOS-a jest što je to sustav na nižoj razini od ostalih, programi i funkcije za obradu prekida su jače povezane sa samim operacijskim sustavom (zahtijevaju posebna znanja programera o načinima rada FreeRTOS-a). Ostali navedeni sustavi imaju znatno odijeljene dijelove programa i jezgre.

13.1.1. Operacijski sustav FreeRTOS

[FreeRTOS] kao i QNX [Neutrino], [VxWorks] te [MicroC/OS-II] je namijenjen ugradbenim računalnim sustavima za rad u stvarnom vremenu i optimiran prema kriteriju malih zahtjeva prema sklopovlju (spremnički prostor i procesorska moć) te se može koristiti i na mikrokontrolerima. Za razliku od prethodno navedenih, izvorni kod FreeRTOS-a je slobodno dostupan (besplatan), kao što mu i samo ime kaže.

Korištenje FreeRTOS-a zahtjeva detaljnije poznavanje njegova sučelja i načina rada. U ovom odjeljku navedena su neka svojstva tog sustava kao i mogućnosti izgradnje programske potpore zasnovane na njemu.

Isključivanje nepotrebnih dijelova jezgre

Radi postizanja što manjeg zahtjeva na spremnički prostor, mnoge se operacije jezgre mogu izostaviti iz prevođenja. Primjerice, ako se sučelje, tj. jezgrina funkcija `Neka_Jezgrina_Funkcija` ne koristi onda u odgovarajućoj konfiguracijskoj datoteci treba postaviti 0 umjesto 1 u liniji (dodati liniju ako ne postoji):

```
#define INCLUDE_Neka_Jezgrina_Funkcija  0
```

Konfiguracijska datoteka je najčešće `FreeRTOSConfig.h` smještena u direktorij s programima koji koriste FreeRTOS. Koje se sve funkcije mogu isključiti iz prevođenja vidi se iz koda

FreeRTOS-a jer su takve funkcije ograđene mehanizmom prikazanim idućim primjerom.

```
#if ( INCLUDE_Neka_Jezgrina_Funkcija == 1 )
    tip Neka_Jezgrina_Funkcija ( parametri )
    {
        ...
    }
#endif  /* INCLUDE_Neka_Jezgrina_Funkcija */
```

Imenovanje varijabli

Imena varijabli i funkcija sadrže prefikse koji označavaju tip varijabli, odnosno povratne vrijednosti funkcije. Tako će cjelobrojna varijabla imati prefiks `x`, pozitivna cjelobrojna varijabla `ux`, kazaljka `p`, tj. `pux` za kazaljku na cjelobrojnu varijablu, `v` za funkciju tipa `void` i slično. Npr. prototip funkcije za stvaranje dretve jest:

```
BaseType_t xTaskCreate (
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    uint16_t usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pvCreatedTask
);
```

Zaštita pri korištenju dijeljenih sredstava sustava

Poprilično detaljno poznavanje rada sustava je neophodno kod FreeRTOS-a. Primjerice, nisu sve operacije koje FreeRTOS pruža implicitno zaštićene kritičnim odsječkom već neke od njih treba dodatno ograditi sinkronizacijskim funkcijama. Jedan takav primjer je korištenje gomile (engl. *heap*) sučeljem `malloc` i `free` koji kojih treba onemogućiti zamjenu s trenutne dretve na neku drugu radi očuvanja konzistentnosti strukture podataka kojom se upravlja gomilom. Jedan od primjera kako to napraviti nalazi se u nastavku.

```
    vTaskSuspendAll ();
    {
        pvReturn = malloc ( xWantedSize );
    }
    xTaskResumeAll ();
```

Navedeni primjer prikazuje potrebu upravljanja sustavom na najnižoj razini u samom programu. Gornji primjer neće prekidati druge dretve, ali prekidi hoće. Zabrana prekida manjeg prioriteta od dretve može se napraviti s `taskENTER_CRITICAL()` (ponovna dozvola s `taskEXIT_CRITICAL()` ili svih prekida s `taskDISABLE_INTERRUPTS()`.

Pokretanje raspoređivača dretvi

Pri pokretanju sustava višedretvenost nije omogućena niti ona to mora biti. Pokretanje višedretvenosti postiže se pozivom `vTaskStartScheduler()`. Primjerice, funkcija koja (nakon inicijalizacije) preuzima kontrolu nad sustavom, stvara potrebne dretve i slično, može izgledati kao u sljedećem primjeru.

```
void vFunkcija ( void )
{
    TaskHandle_t xHandle = NULL;

    /* Inicijalizacija, stvaranje ostalih potrebnih objekata */

    /* Stvaranje bar jedne dretve (prije pokretanja raspoređivača) */
    xTaskCreate (
        vDretva, "IME_DRETVE", VEL_STOGA, NULL, prioritet, &xHandle
    );

    /* ... stvaranje ostalih dretvi i potrebnih objekata */

    /* Pokretanje prioritetnog raspoređivača */
    vTaskStartScheduler ();

    /* Kontrola neće doći ovdje dok se ne zastavi raspoređivač
     * od strane neke dretve pozivom: vTaskEndScheduler ();
     * tj. najčešće se ovdje nikad ne vraća! */

    /* primjer micanja dretve iz sustava */
    if ( xHandle != NULL )
        vTaskDelete ( xHandle );
}
```

Same dretve trebaju biti oblikovane kao funkcije koje nikada ne završavaju odnosno ako trebaju završiti onda one (ili druge dretve) trebaju pozvati `vTaskDelete(pxTask)`. Struktura početne funkcije dretve treba dakle izgledati kao u nastavku.

```
void vDretva ( void * pvParameters )
{
    for (;;)
    {
        /* Posao dretve */
    }
}
```

Manje zahtjevne dretve – niti

Višedretvenost može za neke sustave biti suviše zahtjevna na spremnički prostor jer za svaku dretvu treba opisnik i zaseban stog. Za takve sustave (ali i druge), umjesto pravih dretvi (ili i paralelno s njima) mogu se koristiti jednostavnije dretve, nazovimo ih *nitima* (u FreeRTOS-u su nazvane *co-routine*). Sve niti jedne aplikacije dijele isti stog, funkcije upravljanja su jednostavnije, ali ne mogu se sve jezgirne funkcije koristiti. Zbog zajednička stoga potrebno je posebno paziti i na oblikovanje aplikacije.

Međudretvena sinkronizacija i komunikacija

Od mogućnosti međudretvene komunikacije treba izdvojiti redove poruka, semafore: binarne, opće, binarne s nasljeđivanje prioriteta (izvorno *mutex semaphore*) i alarme. Ako dretva treba čekati na više događaja (poruku ili semafor) mogu se koristiti mehanizmi skupova redova (engl. *queue sets*), gdje se dretva može blokirati dok bar jedan od redova u skupu ne postane prolazan (ima poruku ili semafor postane prolazan).

Pozivi jezgrinih funkcija iz prekidnih funkcija

Većina funkcija za upravljanje dretvama i povezanim objektima ima i inačice pripremljene za poziv iz obrade prekida (engl. *from interrupt service routine*) sa sufiksom `FromISR`. Primjerice, funkcija za slanje poruke u red poruka `xQueueSend` ima alternativnu inačicu `xQueueSendFromISR` za korištenje iz obrade prekida.

```
BaseType_t xQueueSend (
    QueueHandle_t xQueue,
    const void * pvItemToQueue,
    TickType_t xTicksToWait
);
BaseType_t xQueueSendFromISR (
    QueueHandle_t xQueue,
    const void *pvItemToQueue,
    BaseType_t *pxHigherPriorityTaskWoken
);
```

Preko trećeg parametra funkcije `xQueueSendFromISR` dobiva se informacija o tome je li se dodavanjem poruke u red oslobodila neka prioritetnija dretva od trenutno aktivne (i prekinute ovim prekidom), u kom slučaju bi trebalo pozvati raspoređivač. Skica takve prekidne funkcije dana je u nastavku.

```
void vPrekidnaFunkcija(void)
{
    QueueHandle_t xQueue = dohvati_red();
    const void *pvItemToQueue = dohvati_poruku();
    BaseType_t pxHigherPriorityTaskWoken;

    xQueueSendFromISR(xQueue, &pvItemToQueue, &pxHigherPriorityTaskWoken);
    if ( pxHigherPriorityTaskWoken )
    {
        portSAVE_CONTEXT();
        vTaskSwitchContext();
        portRESTORE_CONTEXT();
    }
    asm volatile ( "reti" );
}
```

U primjeru treba primijetiti da se spremanje konteksta i obnova konteksta radi 'ručno' zato što se radi o funkciji koja se zove iz obrade prekida (sama funkcija `vTaskSwitchContext()` samo odabire najprioritetniju za aktivnu dretvu, ali ne radi promjenu konteksta).

Ovisno o sklopovlju na koji se FreeRTOS stavlja obrade prekida mogu se prekidati prioritetnijim prekidima (kada to sklopovlje dozvoljava). U tom slučaju treba pažljivo odabrati prioritete prekida i dretvi i programskih prekida i sklopovskih prekida.

FreeRTOS je operacijski sustav na vrlo niskoj razini upravljanja, ali zato omogućuje ugradnju u sustave s minimalnim spremničkim prostorom (npr. u sustave koji imaju samo 4 KB radnog spremnika!).

13.1.2. Operacijski sustav QNX Neutrino

[Neutrino] je mikrojezgra koja uključuje osnovna sučelja definirana POSIX standardom za ugrađene sustave integrirane s osnovnim sustavom prosljeđivanja poruka QNX. U jezgru su uključeni dijelovi za upravljanje dretvama, sustavom komunikacije porukama, signalima, vremenom i brojilima, prekidima, semaforima, varijablama međusobnog isključivanja i uvjetnim varijablama.

Posebna pažnja pri izgradnji jezgre posvećena je povećanju brzine rada jezgrinih funkcija te smanjenju neprekidivog dijela jezgre. Jezgrine su funkcije zbog toga u većini svog trajanja prekidive. Isto tako prekidima su pridijeljeni prioriteti te se obrada prekida nižeg prioriteta može prekinuti radi obrade prekida većeg prioriteta.

QNX koristi procese (kojima pripadaju dretve). Prioriteti dretvi kreću se od 1 do 63, s time da veći broj označava veći prioritet. U sustavu se razlikuje stvarni prioritet (engl. *real priority*) i trenutni prioritet (engl. *effective priority*) koji dretve mogu imati. Trenutni prioritet uglav-

nom je jednak stvarnom, ali zbog nasljeđivanja prioriteta ili zbog algoritma raspoređivanja on može biti različit. Pripravne dretve raspoređene su u jedan od 64 reda, prema prioritetu. Za izvođenje se uvijek odabire prva dretva iz reda s najvećim prioritetom. U sustavu postoji četiri algoritma raspoređivanja koji se mogu koristiti i koji se definiraju na razini svake pojedine dretve, tj. različite dretve mogu koristiti različite algoritme: raspoređivanje prema redu prispijeća, podjelom vremena, prilagodljivo[1] te sporadično raspoređivanje. Raspoređivači mijenjaju stanja i prioritete dretvi kojima rukuju te ih jezgra dalje raspoređuje isključivo prema prioritetu.

Razmjena poruka osnovni je mehanizam međuprocesne komunikacije koji je naslijeđen iz sustava QNX te je optimiran s obzirom na učinkovitost. Osim njega na raspolaganju su signali, mehanizmi razmjene poruka POSIX, zajednički spremnički prostor i cjevovodi.

13.1.3. Operacijski sustav VxWorks

Među najpoznatije operacijske sustave za rad u stvarnom vremenu spada i [VxWorks]. Monolitna jezgra zavidne brzine, mnoštvo podržanih standarda te moćno razvojno okruženje neki su od uobičajenih atributa koji se vežu uz navedeni sustav. Pored vlastitog sučelja za korištenje u sustavima za rad u stvarnom vremenu, u jezgru je ugrađena podrška i za sučelja POSIX.

Pored pretpostavljenog prioritetnog raspoređivanja, podržano je i raspoređivanje podjelom vremena. Za razliku od većine operacijskih sustava, za opis osnovne jedinice rada koristi se pojam zadatka (engl. *task*) koji zapravo predstavlja dretvu, dok se koncept procesa kao okvira u kojem se dretve izvode ne koristi (u novijim inačicama neki oblik procesa se ipak pojavljuje).

Radi bržeg prihvata prekida, oni se obrađuju izvan konteksta bilo koje dretve. Iz funkcija za obradu prekida mogu se stoga pozivati gotovo sve funkcije jezgre osim onih koje bi mogle uzrokovati zaustavljanje daljnjeg izvođenja, što nije dozvoljeno u obradi prekida.

Za komunikaciju među dretvama postoji nekoliko mehanizama: zajednički spremnički prostor, semafori, redovi poruka, cjevovodi, signali, pozivi udaljenih funkcija i korištenjem mrežnog podsustava.

Rješavanje problema inverzije prioriteta obavlja se protokolom nasljeđivanja prioriteta.

13.1.4. Operacijski sustav Windows EC

Prema navodima proizvođača, porodica operacijskih sustava Windows Embedded Compact [Windows EC] (prijašnje ime Windows CE) izgrađena je tako da zadovoljava široko područje inteligentnih uređaja. Zbog sličnosti arhitekture jezgre i njenog sučelja s ostalim Win32 arhitekturama, sustav ima potencijalno vrlo veliki broj programera koji uz malo dodatnog napora mogu izgrađivati rješenja zasnivana na ovom operacijskom sustavu. Pored poznate arhitekture jedna od prednosti ovog sustava su razvojni alati.

Jezgra sustava pruža osnovnu funkcionalnost operacijskog sustava u upravljanju procesima, dretvama i spremničkim prostorom za koje se koristi straničenje. Odnos procesa i dretvi jednak je kao i na ostalim Win32 sustavima, tj. jedan proces sadržava jednu ili više dretvi koje dijele isti adresni prostor te mogu učinkovitije raditi na istom zadatku. Prioriteti dretvi koji se koriste u procesu raspoređivanja su u području od 0 do 255, gdje manji broj predstavlja veći prioritet. Prioriteti su podijeljeni u nekoliko skupina te je za svaku predviđena posebna namjena. Koristi se kružno raspoređivanje temeljeno na prioritetima. Pripravne dretve najvećeg prioriteta dijele procesorsko vrijeme tako da svaka dobije po kvant vremena. Nakon što raspoređene dretve

[1] Prilagodljivo raspoređivanje (engl. *adaptive scheduling*) je vrlo slično raspoređivanju podjelom vremena, uz razliku da se dretvi smanjuje prioritet za jedan na kraju njena kvanta vremena. Prioritet se takvoj dretvi može smanjiti samo za jedan ukupno, tj. ako je prioritet već smanjen za jedan na kraju jednog kvanta vremena, na kraju drugog kvanta dodijeljenog dretvi prioritet ostaje isti (onaj već smanjen za jedan u odnosu na početni prioritet). Ako se dretva zaustavi prije isteka kvanta vremena prioritet joj se vraća na početnu vrijednost.

završe s radom ili više nisu u redu pripravnih, s izvođenjem nastavljaju dretve sljedećeg nižeg prioriteta. Problem inverzije prioriteta rješava se protokolom nasljeđivanja prioriteta.

Prihvat i obrada prekida podijeljena je na dva dijela: jezgrina prekidna rutina (engl. *kernel-mode interrupt service routine – ISR*) i dretva za obradu prekida (engl. *user-mode interrupt service thread – IST*). ISR se obavlja unutar nadglednog načina rada i namijenjen je obavljanju kraćih poslova pri pojavi prekida. Sučelje preko kojega ISR treba koristiti jezgru omogućuje da se nakon završetka rutine ISR, ako je potrebno, pokreće IST koja obavlja ostatak posla koji može potrajati te se zato izvodi izvan jezgre u korisničkom načinu rada. Za različite prekide mogu postojati različite rutine ISR i dretve IST.

13.1.5. Operacijski sustav Android

U studenom 2007. osnovana je Open Handset Alliance, udruženje nekoliko tvrtki među kojima se nalaze: Texas Instruments, Broadcom Corporation, Google, HTC, Intel, LG, Marvell Technology Group, Motorola, Nvidia, Qualcomm, Samsung Electronics, Sprint Nextel i T-Mobile. Cilj udruge je uvođenje standarda za mobilne uređaje. U vrijeme nastanka, organizacija je napravila svoj prvi proizvod, Android, platformu za mobilne uređaje napravljenu na temelju jezgre Linux. U početku izvorni kôd Androida nije bio u potpunosti otvoren. Međutim, zbog kritika i pritiska ostalih kompanija, Google je 21. listopada 2008. u potpunosti otvorio kôd. Od tada je Android prva besplatna i potpuno prilagodljiva mobilna platforma.

Android je temeljen na Linuxu, ali ga se ne smije poistovjećivati s Linuxom. Za razliku od Linuxa, Android nema istu podršku za rad u grafičkom okruženju (engl. *native windowing system*). Također, nema glibc paket koji sadrži standardne biblioteke programskog jezika C korištene u Linuxu. Google se odlučio za Linux radi njegovog upravljanja spremnikom i procesima, upravljačkih programa i otvorenog kôda. Ostatak sustava je ipak posebno osmišljen i ostvaren tako da zadovoljava predviđenu okolinu u kojoj će se koristiti.

Svi programi za Android se pripremaju u Javi te se izvode na virtualnom stroju Dalvik. Odlike Dalvika su višezadaćni rad, učinkovito iskorištavanje sredstava sustava i visoko optimirana interpretacija strojnog koda.

Radi otvorenog koda (tj. prenosivosti na razne uređaje), visoke optimiranosti, jednostavnosti i stalnih nadogradnji, Android koriste mnogi mobilni uređaji, većinom telefoni i ručna računala (engl. *tablet*). S obzirom na njegovu rasprostranjenost, činjenicu da se i dalje razvija, da za njega postoji integrirano tržište programa, Andorid će vrlo vjerojatno opstati dugo u budućnosti.

13.1.6. Operacijski sustav iOS

Operacijski sustav iOS nalazi se na uređajima tvrtke Apple Inc. Pojavio se s prvim iPhone telefonom 2007. i od tada se koristi na njihovim uređajima (iPhone, iPod Touch, iPad i Apple TV). Opercijski sustav je nastao na temelju OS X sustava koji se koristi na Appleovim računalima, a koji je temeljen Darwin projektu.

iOS je posebno izgrađen za pametne telefone, ručna računala i multimedijalne uređaje te su neki elementi operacijskog sustava malo drukčiji od uobičajenih. Primjerice, višedretvenost postoji, ali je prilagođena za namjenu uređaja. Ako program nije u fokusu, on može nastaviti rad "u pozadini" samo ako se radi o posebnim operacijama kojima se pristupa preko unaprijed definiranih sučelja (glazba, video i slično).

Slično kao i Android, iOS se i dalje razvija, ima razvijenu podršku za kupovinu i prodaju programa preko Interneta. Za razliku od Androida, iOS je zatvoren, njegov izvorni kôd nije dostupan i pokreće se samo na uređajima koje Apple proizvodi (i osmišljava).

13.2. Svojstva različitih tipova operacijskih sustava

Odabir operacijskog sustava za pojedine primjene treba obaviti sukladno očekivanjima funkcionalnosti i usklađenosti s okolinom u koju se ugrađuje. Također treba planirati mogućnosti ažuriranja i nadogradnje kroz predviđeni životni vijek sustava. Cijena nabavke i podrške proizvođača pri održavanju trebaju također biti jedan od razloga pri odabiru (za "dugovječne sustave").

Slika 13.1. prikazuje odnos svojstava različitih kategorija operacijskih sustava.

Slika 13.1. Odnos svojstava između općih sustava i onih posebne namjene

Operacijski sustavi opće namjene građeni su generički, kao i samo sklopovlje za takva računala. Oni su napravljeni da mogu raditi mnoštvo stvari uz zadovoljavajuću kvalitetu i nisku cijenu. Promatrajući svojstva takvih sustava u kontekstu uporabe za upravljanje vremenski kritičnih procesa uočavaju se neki nedostaci koji su posljedica otvorenosti operacijskih sustava opće namjene raznim sklopovskim i programskim rješenjima. Zato je odziv takvih sustava vrlo nepredvidiv, često neprihvatljivo dug, a kako bi se takvi sustavi mogli koristiti za upravljanje procesa sa strogim vremenskim ograničenjima. Za nekritične elemente sustava, gdje su zadana blaža vremenska ograničenja, moguće je odabrati i operacijske sustave za opću uporabu. Pravilnim podešavanjem sustava može se znatno poboljšati ponašanje sustava u smislu pouzdanosti i predvidivosti ponašanja.

S druge strane, operacijski sustavi posebno osmišljeni za sustave sa strogim vremenskim ograničenjima, kao što su to operacijski sustavi za rad u stvarnom vremenu te operacijski sustavi za ugrađena računala, imaju, naravno, znatno bolja vremenska svojstva. Nedostatak takvih sustava je u manjoj podršci sklopovlju i dobavljivosti gotovih programskih rješenja.

Između pravih operacijskih sustava za rad u stvarnom vremenu i operacijskih sustava opće namjene postoji područje u kojemu se nalaze prilagođeni operacijski sustavi opće namjene, ali značajno boljih svojstava od početnih sustava. Prednosti takvih sustava su u dostupnosti gotovo jednakih razvojnih alata i podržanih protokola kao i u sustavima opće namjene.

13.2.1. Operacijski sustavi za rad u stvarnom vremenu

Primjere zašto neki operacijski sustavi opće namjene nisu pogodni za kritične sustave može se potražiti iz raznih studija. Jedna od takvih uspoređuje eksperimentalne rezultate provedene na Linuxu s jezgrama 2.4 i 2.6 te s prilagođenim Linuxom za uporabu u SRSV-ovima. U eksperimentu su mjerena kašnjenja od trenutka pojave događaja do početka njegove obrade (engl. *preemption latency*). Tablica 13.1. prikazuje rezultate jednog eksperimenta. Prosječna kašnjenje su samo malo veća kod običnih sustava, međutim, kada se pogledaju svi slučajevi i izvuku najgori slučajevi onda se jasno vidi da su operacijski sustavi opće namjene (koji nisu prilagođeni) značajno lošiji.

Tablica 13.1. Izmjerena kašnjenja do početka obrade prekida u mikrosekundama ([Laurich, 2004])

	2.4 Linux	2.6 Linux	2.4 Linux RTAI	2.4 Linux LXRT
Prosječna kašnjenja	12 – 14	12 – 14	8 – 10	10 – 12
Najveća kašnjenja	4446	578	42	50

Novija istraživanja samo potvrđuju podatke od zadnjih 15ak godina. Slika 13.2. prikazuje vremena odziva na novijem Linuxu (inačica jezgre 2.6.24) i njegovoj inačici za SRSV.

Legenda: svaki znak "+" označava jedan događaj u običnom Linuxu dok znak "×" događaj u prilagođenom Linuxu (SRSV inačici); ordinata predstavlja vrijeme odziva, a apscisa redni broj događaja

Slika 13.2. Usporedba vremena odziva na Linuxu ([Williams, 2008])

Kao što se vidi iz slike 13.2., vrijeme odziva na događaje je znatno kraće za prilagođenu inačicu Linuxa i ne prelazi 50 μs. S druge strane, standardni Linux, iako u prosjeku događaje obrađuje s kašnjenjem od oko 50 μs ima značajan broj odziva čija su kašnjenja za događajem veća i od nekoliko milisekundi!

Takvi ispadi, tj. produljena kašnjenja u početku obrade, premda vrlo rijetka, za mnoge su sustave neprihvatljiva. S druge strane, kod drugih sustava rijetka kašnjenja u obradi događaja ne moraju nužno označavati katastrofalnu grešku, već možda samo smanjenje kvalitete ili se

njihov utjecaj može i zanemariti.

Ako je zaista potrebno, neki elementi sustava moraju se izvoditi na nekom od operacijskih sustava za rad u stvarnom vremenu. Svojstva takvih sustava koji se danas mogu pronaći na tržištu su približno jednaka. Ono što se može razlikovati jest podrška prema podržanom sklopovlju, alatima za razvoj programa, skupu podržanih protokola i standarda te podršci koju proizvođač pruža pri razvoju. Minimum koji treba tražiti jest podrška standardnim sučeljima POSIX za rad u stvarnom vremenu kako bi se isti programi u budućnosti mogli jednostavnije prenijeti na druge platforme.

13.2.2. Operacijski sustavi opće namjene

Operacijski sustavi opće namjene mogu biti dostatni za nekritične elemente sustava. Npr. novije inačice Windows operacijskih sustava, od XP, 2003, Viste, 2008, 7, 8, 10 itd. Prednosti korištenja tih sustava su u dostupnosti svih tehnologija, protokola, standarda i alata za njih, kao i mogućnosti njihovog korištenja na svakom osobnom računalu. Odabir jednog od njih i njihova međusobna sukladnost i mogućnost suradnje pruža sigurnost u nastavak razvoja i održavanja (s obzirom na to da su ti sustavi zastupljeni u velikoj većini današnjih računalnih sustava).

"Alternativno" rješenje može biti korištenje sustava zasnovanog na Linuxu. Linux u današnje vrijeme razvija dobrovoljna zajednica, ali često uz pomoć većih tvrtki (primjerice Google). Iako su besplatni za korištenje, sustavi temeljni na Linuxu su (barem) usporedivi s komercijalnim sustavima u pogledu učinkovitosti, dostupnosti programa i alata za razvoj. Kao i za operacijski sustav Windows tako se i za Linux može predvidjeti da će se u bližoj budućnosti nastaviti razvijati.

Iako značajno lošijih svojstava od operacijskih sustava za rad u stvarnom vremenu, operacijski sustavi opće namjene se ipak mogu primijeniti u velikom broju slučajeva. Napretkom tehnologije današnja su računala vrlo brza pa se nedostaci u nedeterminizmu često (ali ne uvijek!) mogu nadomjestiti brzinom rada za sustave s blažim vremenskim ograničenjima.

13.2.3. Prilagođeni operacijski sustavi opće namjene

U području između operacijskih sustava za rad u stvarnom vremenu i onih opće namjene može se izdvojiti nekoliko značajnijih.

Real-Time Linux [RT Linux] (Linux s `CONFIG_PREEMPT_RT` dodatakom, engl. *patch*) jedan je od njih. Navedeni dodatak mijenja neke jezgrine funkcije, tj. omogućuje njihovo prekidanje u izvođenju. Jezgra tog sustava je stoga znatno predvidljivija i moguća kašnjenja su poprilično smanjena pa se koristi u mnogim sustavima za rad u stvarnom vremenu s ublaženim vremenskim ograničenjima. Osim njega koji je besplatno dostupan, razne tvrtke su pristupile modifikaciji Linuxa te prodaju svoje modifikacije. Primjeri komercijalno dostupnih prilagođenih Linuxa su: RTLinux (FSMLabs, Windriver), Lineo – Embedix Realtime, REDSonic – REDICE Linux, MonaVista Real Time Extensions, LynuxWorks – BlueCat RT i TimeSys – Linux/Real-Time+. Uobičajeno rješenje koje koristi većina navedenih sustava je korištenje vlastite jezgre u kojoj se sam Linux izvodi kao zaseban proces.

Primjerice, RTLinux [RTLinux] ima svoju jezgru koja izvodi kritične zadatke upravljanja (koji se pripremaju za tu jezgru). Linux je u tom sustavu samo jedan proces i to proces s najmanjim prioritetom. Svi programi pripremljeni za jezgru imaju veći prioritet od Linuxa, koji zapravo jezgru vidi kao sklopovlje (jezgra "maskira" sklopovlje pravom Linuxu). Komunikacija zadataka kojima upravlja RTLinux i zadataka kojima upravlja Linux može se odvijati preko zajedničkog spremnika te preko struktura FIFO. I jedni i drugi se u Linuxu vide kao uređaji, dok ih dretve u RTLinuxu koriste posebnim sučeljem. Navedeno prikazuje slika 13.3.

Microsoft neke svoje opće operacijske sustave nudi i u posebnoj izvedbi u kojoj je moguće odre-

Slika 13.3. RTLinux arhitektura

diti (pri postavljanju sustava) koji su elementi potrebni, a koji nisu. Na ovaj način se može napraviti znatno predvidljiviji sustav od standardnih operacijskih sustava izostavljajući u potpunosti nepotrebne komponente. Windows Embedded Standard 7 je jedan takav proizvod. On je potpuno sukladan s Win32 standardom, tj. svi programi koje rade na Windows platformama radit će i na ovom operacijskom sustavu. Standardni servisi i protokoli sadržani su i u ovim sustavima, kao i podrška za tehnologiju .NET.

13.2.4. Odabir operacijskih sustava

Analizom potreba za pojedine probleme treba odabrati prikladan operacijski sustav. Ako su vremenska ograničenja vrlo stroga, za taj dio sustava potrebno je odabrati prikladan OS za SRSV. Za manje kritične komponente poželjno je ipak odabrati neki opći operacijski sustavi (Windows ili Linux) ili njihove modifikacije (npr. Real-Time Linux ili Windows Embedded Standard 7) zbog veće dostupnosti razvojnih alata, podrške raznim tehnologijama i protokolima, kao i predviđenom životnom vijeku, njihovu daljem razvoju i podršci. Odabirom ovakvih operacijskih sustava s kontinuiranim razvojem i ogromnom korisničkom bazom maksimizira se rok podrške za operacijske sustave pa tako i posredno podršku za većinu novih tehnologija koje će se pojaviti u skoroj budućnosti, koji bi mogli postati potrebni u procesu unaprjeđenja sustava.

Korištenjem standardnih tehnologija podržanih od strane više proizvođača omogućuje se modularna izgradnja sustava, koji po sastavu (sklopovski i programski) može biti i heterogen. Odluka o odabiru operacijskog sustava za pojedine elemente sustava u takvom slučaju ne mora biti konačna, već se on može promijeniti i naknadno bez značajnijeg dodatnog truda u prilagodbi aplikacija za taj sustav.

Konačno, ponekad ipak može biti potrebno izgraditi vlastiti sustav jer postojeći iz raznih razloga ne odgovaraju (cijenom, svojstvima, zahtjevima na sklopovlje, ...).

Pitanja za vježbu 13

1. Navesti nekoliko operacijskih sustava za korištenje u SRSV-ovima sa strogim vremenskim ograničenjima te neke za sustave s blažim vremenskim ograničenjima.

2. Koja svojstva ograničavaju uporabu operacijskog sustava u ugrađenim sustavima?
3. Što su to “prilagođeni” operacijski sustavi? Kako se ostvaruju? Prikažite na primjeru operacijska sustava RTLinux.
4. Koji su mogući izbori pri izgradnji jednostavnijih ugrađenih sustava, a koji pri izgradnji složenijih sustava?

Dodaci

Dodatak A - Ostvarenje za arhitekturu ARM

A.1. Obilježja procesora ARM

procesor ARM je pravi predstavnik procesora za ugradbene sustave, iako se u novije vrijeme koristi i u "jačim sustavima" (npr. pametnim telefonima i tabletima). ARM je procesor s ograničenim skupom instrukcija (engl. *RISC – Reduced Instruction Set Computer*). Duljina procesorske riječi kod procesora ARM je 32-bita iako neki noviji modeli procesora imaju 64-bitovnu riječ. Radi povećanja gustoće instrukcija (smanjenje potrebne veličine programa) neki ARM procesori podržavaju i sažete načine rada s instrukcijama duljine 16 bita (*Thumb* način rada).

Programski model procesora ARM sastoji se od skupa instrukcija i registara dostupnih procesoru. Na raspolaganju je 16 registara opće namjene, `r0-r15`, s time da se neki od tih registara koriste za posebne namjene. `r15` predstavlja programsko brojilo i često se označava s `pc` (engl. *program counter*). `r13` se koristi za rad sa stogom te se za njega koristi i oznaka `sp` (engl. *stack pointer*), dok `r14` služi kao registar za privremenu pohranu sadržaja programskog brojila pri pozivu potprograma ili pojavi prekida te ima i dodatnu oznaku `lr` (engl. *link register*). Uz navedene registre opće uporabe postoji još i statusni registar `cpsr/spsr` (engl. *current/saved program status register*). Neki od navedenih registara različiti su za različite načine rada procesora.

procesor ARM može biti u jednom od 6 načina rada, koji su određeni vrijednostima zadnjih pet bitova statusnog registra:

- korisnički način rada (engl. *user mode*)
- sustavski način rada (engl. *system mode – sys*)
- prekidni načina rada (engl. *interrupt mode – irq*)
- brzi prekidni načina rada (engl. *fast interrupt mode – fiq*)
- način rada u slučaju nedefinirane instrukcije (engl. *undefined mode – und*)
- način rada u slučaju nemogućnosti dohvata operandi (engl. *abort mode – abt*)
- način rada za obradu programskih prekida (engl. *software interrupt – svc*).

Od registara opće namjene neki su registri različiti za pojedine načine rada (osim za sustavski način rada koji koristi registre korisničkog načina). Tako se npr. pri prijelazu iz korisničkog načina rada u prekidni način rada (*irq*) umjesto dosadašnjih registara `r13`, `r14` i statusnog registra koriste istoimeni registri ali ovog načina rada, tj. registri `r13_irq`, `r14_irq` te `spsr_irq` (u instrukcijama dohvatljivi na isti način, bez `_irq` dodatka). Za svaki od prekidnih načina rada procesor ARM ima dodatne registre (`r13`, `r14` i `cpsr`) koje aktivira pri prijelazu u taj način rada (deaktivira korisničke registre i aktivira prekidne registre).

Promjena načina rada obavlja se pri pojavi prekida ili izravnim mijenjanjem određenih zastavica u statusnom registru koje je moguće napraviti iz svih načina rada osim iz korisničkog. Ponašanje procesora pri pojavi prekida može se opisati u četiri koraka:

1. zabranjuje se daljnje prekidanje
2. procesor se prebacuje u novi način rada
3. programsko brojilo se pohranjuje u registar `r14_mode` i `cpsr` u `spsr_mode`, gdje je `mode` oznaka prekida
4. u programsko brojilo upisuje se vrijednost između 00_{16} i $1C_{16}$, ovisno o prekidu.

Promjena načina rada iz korisničkog u prekidni ilustrirana je slikom A.1.

Slika A.1. Promjena načina rada prihvatom `irq` prekida

Brzi prekidi imaju zadnju adresu u tablici te je moguće da kôd za obradu slijedi odmah od te adrese, bez grananja. Također, brzi prekidi imaju na raspolaganju više privatnih registara od ostalih načina rada (uz `r13`, `r14` i `cpsr` imaju i zasebne `r8-r12`) ciljajući na poboljšanje učinkovitosti pri obradi takvih prekida. U jednostavnijim slučajevima ti će registri biti dovoljni za obradu prekida te se neće trošiti vrijeme na spremanje konteksta, tj. sadržaja registara.

Pri povratku iz obrade prekida potrebno je vratiti procesor u stanje koje je bilo prije pojave prekida, tj. potrebno je vratiti stanje svih registara. Vrijednost programskog brojila te statusni registar treba vratiti u istoj operaciji te za to postoje posebne inačice instrukcija bilo da je povratna adresa spremljena na stogu ili u registru `r14`. Pri povratku iz nekih prekida potrebno je najprije korigirati vrijednost programskog brojila (zbog protočne arhitekture procesora). Pri završetku obrade prekida i brzih prekida vrijednost programskog brojila treba umanjiti za 4 dok za prekide dohvata za 8.

Zbog mogućnosti pojave više prekida u isto vrijeme pojedinim je prekidima pridijeljen različiti prioritet i to redom: reset (najveći prioritet), greška pri dohvata podataka iz spremnika, brzi prekid, prekid, greška pri dohvatu instrukcije (ili njenih operandi), programski prekidi i nedefinirana instrukcija.

Za pristup i upravljanje ulazno izlaznim napravama koristi se mapiranje adresa spremničkih lokacija te prekidima. Registri ulazno-izlaznih naprava dostupni su na određenim adresama adresnog prostora.

Način pohrane registara u spremnik može biti u *big-endian* ili *little-endian* načinu, tj. podržana su oba načina i odabir se obavlja postavljanjem određenih premosnika na samom sklopu.

A.2. Korištena arhitektura ARM

Prilagodba sloja arhitekture za procesore ARM napravljena je korištenjem sklopovlja *Versatile PB926EJ-S* [ARM926EJ-S] prikazano na slici A.2.

Slika A.2. ***Versatile PB926EJ-S*** **razvojno okruženje**

Navedeno sklopovlje je razvojno, s mnoštvo naprava od kojih je samo jedan mali dio korišten u Benu. Zapravo korišteni su samo elementi: procesor (ARM926EJ-S), radni spremnik, serijski port UART0 (preko sklopa PL011), upravljač prekidima PL190 te brojila 0 i 1 sklopa SP804 za ostvarenje satnog podsustava.

Kao razvojni alat za izgradnju sustava korišten je paket *Sourcery CodeBench Lite Edition for ARM EABI* [ARM-EABI], a za emulaciju ARM sustava te pokretanje izgrađenog sustava opet je korišten QEMU za ARM (`qemu-system-arm -M versatilepb ...`). S obzirom na to da u ovom slučaju na arhitekturi x86 (na Linuxu) izgrađuje program za ARM, za ovakav skup alata (koji se na jednoj arhitekturi koristi za pripremu programa za drugu arhitekturu) koristi se engleski naziv *cross compiler*.

S obzirom na to da procesor ARM ima nekoliko načina rada, kao osnovni način rada odabran je sustavski način rada u kojemu su dostupne sve operacije. Korisnički način rada prikazan je jedino u poglavlju gdje se on želi pokazati, tj. u poglavlju `Chapter_08_Processes/02_User_mode`. Od prekidnih načina rada za sve prekide vanjskih naprava korišten je prekidni način (*irq*) i to samo vrlo kratko pri prihvatu prekida. Po prihvatu prekida sustav se ponovno prebacuje u sustavski način rada i u njemu obrađuje prekid.

A.3. Posebnosti u ostvarenju sloja arhitekture

U ovom su odjeljku opisane posebnosti ostvarenja u pojedinim koracima Benua.

A.3.1. Ispis na konzolu

Za razliku od i386 sustava, kod ARM-a se ne koristi zaslon (iako *Versatile PB926EJ-S* ima mali tekstovni LCD zaslon) već se koristi serijska veza. Razlog je u jednostavnosti korištenja serijske veze i njenog povezivanja s emulatorom QEMU.

Ispis preko serije obavlja se jednostavnim upisom znaka u podatkovni registar pristupnog sklopa (PL011) na zadanoj adresi (`UART0_DR`), uz prethodnu provjeru spremnosti za prihvat takvog znaka. QEMU se pokreće tako da sve što dobije sa serijske veze ispiše na zaslon, a sve što primi na konzoli proslijedi na serijsku vezu emuliranog računala (zastavica `-serial stdio` za QEMU).

A.3.2. Prihvat prekida

Kao što je već opisano, pri prihvatu prekida u programsko brojilo se učitava adresa od 0_{16} do $1C_{16}$, ovisno o izvoru prekida. Primjerice, na prekidni signal RESET u programsko brojilo se učitava 0, na *irq* izvor 18_{16} i slično. Da bi se te prekide obradilo zasebnim kodom, na te adrese treba postaviti instrukcije grananja prema potrebnim funkcijama za obradu. Međutim, budući da se sustav učitava drugim programima u radni spremnik (npr. QEMU ili GRUB), postavljanje takvih instrukcija treba napraviti po pokretanju. Navedeno izvodi funkcija `arch_processor_init` u `interrupt.S` koja se prvo poziva iz `arch_startup` (`startup.S`).

Isječak kôda A.1. Chapter_03_Interrupts/01_Exceptions/arch/arm/interrupt.S

```
20 /* interrupt table copied to address 0 */
21 processor_interrupt_table_start:
22         ldr     pc, reset_handler_addr
23         ldr     pc, undef_handler_addr
24         ldr     pc, swi_handler_addr
25         ldr     pc, prefetch_abort_handler_addr
26         ldr     pc, data_abort_handler_addr
27         b       .
28         ldr     pc, irq_handler_addr
29         ldr     pc, fiq_handler_addr
30
31 reset_handler_addr:             .word arch_processor_init
32 undef_handler_addr:             .word arch_int_hndl
33 swi_handler_addr:               .word arch_int_hndl
34 prefetch_abort_handler_addr:    .word arch_int_hndl
35 data_abort_handler_addr:        .word arch_int_hndl
36 irq_handler_addr:               .word arch_int_hndl
37 fiq_handler_addr:               .word arch_int_hndl
38 processor_interrupt_table_end:
39
40
41 arch_processor_init:
42         /* set system mode, set stack */
43         msr     cpsr, #(CPSR_MODE_SYS|CPSR_IRQ)
44         ldr     sp, _system_stack_
45
46         /* stay in system mode until interrupt */
47
48         /* copy vector table to address 0 */
49         mov     r0, #0
50         ldr     r1, =processor_interrupt_table_start
51         ldr     r2, =processor_interrupt_table_end
52 1:      ldr     r3,[r1], #4
```

```
53          str     r3,[r0], #4
54          cmp     r1,r2
55          blt     1b
56
57          /* jump to main */
58          b       arch_jump_to_kernel
59
60
61  _system_stack_:         .word   system_stack + STACK_SIZE
```

U Benu se, radi jednostavnosti i sažetosti prikaza (kao i kod ostvarenja za i386), na svaki prekid poziva ista funkcija `arch_int_hndl`.

Isječak kôda A.2. Chapter_03_Interrupts/01_Exceptions/arch/arm/interrupt.S

```
68  arch_int_hndl:
69          ldr     sp, _irq_stack_         /* set interrupt stack for int. mode */
70          stmfd   sp, {sp}^               /* save thread stack pointer */
71          ldr     sp, [sp, #-4]           /* activate thread stack */
72          stmfd   sp, {r0-r14}^           /* save thread context on its stack */
73          sub     sp, sp, #(15*4)
74          push    {lr}                    /* interrupt return address */
75
76          mrs     r0, cpsr
77          mrs     r1, spsr
78
79          push    {r0-r1}         /* intsrc, cpsr, spsr */
80
81          mov     r2, sp          /* save stack pointer before mode change */
82
83          msr     cpsr, #CPSR_MODE_SYS_IF         /* change to system mode */
84          mov     sp, r2          /* restore stack pointer after mode change */
85
86          /* jump to interrupt handler(r0=cpsr) */
87          bl      arch_interrupt_handler
88
89          pop     {r0-r1}
90
91          mov     r2, sp
92          msr     cpsr, r0        /* change to interrupt handler mode */
93          msr     spsr, r1
94          mov     sp, r2
95
96          pop     {lr}
97
98          and     r0, #0x1f               /* leave only mode bits */
99          cmp     r0, #CPSR_MODE_SVC      /* is it SVC interrupt? */
100
101         ldmfd   sp, {r0-r14}^   /* restore thread context(incl. stack p.) */
102
103         beq     ret_from_svc
104
105 ret_from_irq:
106         subs    pc, lr, #4              /* return from: irq, fiq...*/
107
108 ret_from_svc:
109         subs    pc, lr, #0              /* return from: svc */
```

Uzrok prekida se doznaje preko registra stanja u kojem zadnjih pet bitova određuju način rada procesora što ujedno određuje i uzrok prekida. Uzrok se naknadno gleda u funkciji `arch_interrupt_handler` (`interupt.c`).

I za obradu prekida odabran je sustavski način rada. Međutim, prije prebacivanja u taj način rada potrebno je zapamtiti uzrok prekida koji je neizravno zapisan u registru stanja te povratnu adresu koja je zapisana u `lr`. Stoga se najprije obnavlja kazaljka stoga prekinute dretve (jedine

za sada) te na taj stog pohranjuju redom: svi registri prekinute dretve (`r0-r14`), povratna adresa (adresa kuda se vratiti nakon obrade prekida – sadržaj `lr` trenutnog načina rada) te prijašnji i sadašnji registar stanja (`spsr`, `cpsr`). Tek se tada radi promjena načina rada i poziv funkcije koja će proslijediti obradu prekida na odgovarajuću funkciju.

Povratak iz prekida obavlja operacije obrnutim redoslijedom. Zatečena vrijednost programskog brojila u trenutku prekida može biti i veća od adrese iduće instrukcije (zbog protočne strukture izvedbe procesora). Stoga je prije povratka iz prekida potrebno prilagoditi spremljenu adresu.

Pri obradi prekida se koristi isti stog kao i prije obrade (stog iste dretve). Navedeni način (korištenje samo jednog stoga te sustavski način rada u obradi prekida) odabran je da ARM ostvarenje bude što bliže ostvarenju prihvata i obradi prekida kao kod arhitekture i386. Druge mogućnosti su da procesor ostane u načinu rada u kojem je zatečen po prihvatu prekida te da koristi isti stog kao i sada ili pak svoj vlastiti (za svaki uzrok prekida zaseban).

A.3.3. Korištenje brojila za upravljanje vremenom

Za upravljanje vremenom koristi se brojilo 0 sklopa SP804. Od mogućih načina rada odabrano je odbrojavanje u 32 bita frekvencijom od 1 MHz (pretpostavljena frekvencija za ovaj sustav). Brojilo odbrojava od zadane vrijednosti do nule. Pri dostizanju nule sklop izaziva prekid, učitava prethodno zadanu vrijednost i nastavlja s odbrojavanjem. U obradi prekida jedino je potrebno dojaviti sklopu da je njegov prekid prihvaćen. Stoga se pri prihvati prekida najprije radi upravo to (uklanja njegov zahtjev za prekid), a tek potom obradu prosljeđuje na više razine (u funkciju `arch_timer_handler` u `arch/arm/time.c`).

A.3.4. Serijska veza u oba smjera

Upravljački program za serijsku vezu preko sklopa PL011 ostvaren je na sličan načina kao i za UART kod arhitekture i386, korištenje dodatnih međuspremnika za privremenu pohranu podataka i pri prihvatu novih podataka sa serijske veze i pri slanju podataka. Također, samo je osnovni način rada sklopa podržan, tj. nisu ispitani svi mogući uzroci prekida i stanja prijenosnog kanala.

A.3.5. Višedretvenost

Operacije spremanja i obnove konteksta dretve potrebne za višedretvenost ostvarene su u `arch/arm/context.S` na sličan način kao i kod arhitekture i386.

Isječak kôda A.3. Chapter_07_Threads/02_Threads/arch/arm/context.S

```
10 arch_switch_to_thread:
11         /* r0 = from, r1 = to */
12         cmp     r0, #0          /* is "from" given? */
13         beq     restore
14
15         /* save context of current thread */
16         sub     sp, sp, #4      /* pc will be saved here */
17
18         push    {r0-r12,lr}     /* registers */
19
20         ldr     r5, =return     /* return there */
21         str     r5, [sp, #56]   /* store it below 14 regs */
22
23         mrs     r5, cpsr
24         push    {r5}            /* cpsr */
25
26         str     sp, [r0]        /* save stack address to from->context */
27
28 restore:
```

```
29          ldr     r1, [r1]        /* load stack address from to->context */
30          mov     sp, r1          /* restore stack <= to*/
31
32          pop     {r5}
33          msr     cpsr, r5        /* restore cpsr */
34
35          pop     {r0-r12,lr}     /* registers */
36
37          pop     {pc}            /* 1st time=thread func,
38                                     other times: addr. of next instr.(2) */
39
40  return:
41          mov     pc, lr          /* return from function */
```

Za ostvarenje zamjene konteksta odabran je "čisti" asembler, a ne kao kod arhitekture i386 gdje je to ugrađeno u C datoteku. Razlog je problem kod korištenja stoga i dohvaćanja parametara funkcije. S obzirom na to da kontekst dretve "ručno" sprema na stog (dodanim asemblerskim instrukcijama), kazaljka stoga se povećava te reference na parametre funkcije koje su izražene relativno prema kazaljci stoga više nisu ispravne.

A.3.6. Poziv jezgri preko prekida

Promjena u sloju arhitekture izazvana promjenom načina poziva jezgrenih funkcija vidljiva je u prihvatu prekida, ali i u formatu konteksta dretvi s obzirom na to da se od sada dretve mijenjaju isključivo u jezgri mehanizmom prekida (sada posebnom načinu rada sa zasebnim stogom). Promjene su stoga vidljive u `Chapter_08_Processes/01_Syscall/arch/arm` direktoriju u datotekama `context.h`, `interrupt.S` i `syscall.h` dok se `context.S` sada još jedino koristi za dretve upravljane izvan jezgre (`user_threads` program).

Uobičajena konvencija za prijenos parametara u potprograma kod procesora ARM (za programski jezik C) jest korištenje registara procesora za prijenos prva 4 parametra te stog za ostale. Poziv jezgrene funkcije ostvaren je pozivom funkcije `syscall` (`arm/syscall.S`) koja potom odmah izaziva programski prekid. U trenutku prije izazivanja prekida parametri se nalaze u registrima `r0-r3` te na stogu (od 5. parametra na dalje). U prihvatu prekida pohranjuju se svi registri procesora na stog, ali najprije upravo registri `r0-r3` da bi bili smješteni uz ostale parametre (kada se radi o programskom prekidu). Stoga, bez obzira na broj parametara oni su nakon poziva jezgrene funkcije kompaktno smješteni na stogu. Slična je situacija bila i kod arhitekture i386 kod koje je uobičajena konvencija da se parametri i inače u funkciju šalju putem stoga te ove manipulacije nije bilo potrebno raditi (pogledati poglavlje 12.2.).

A.3.7. Korisnični način rada

Korištenje korisničkog načina rada umjesto sustavskog za izvođenje dretvi zahtjeva samo jednu promjenu – promjenu početnog registra stanja za novu dretvu u kojem će biti postavljen korisnički način rada. Razlog takve sitne izmjene jest u tome što se do sada (do ovog koraka) za sve koristio sustavski način rada koji koristi iste registre procesora kao i korisnički. S obzirom na to da prihvatom prekida iz korisničkog načina se prelazi u prekidni a potom u sustavski, razlika je samo u početku (korisnički način rada u prekidni umjesto sustavski način rada u prekidni).

A.3.8. Ostvarenje procesa

Ostvarenje procesa nije ostvareno za platformu ARM iz nekoliko razloga.

Prvi je razlog u potrebi korištenja modula radi odvojenog prevođenja jezgre i programa. Dok se za arhitekturu i386 može koristiti GRUB za pokretanje, koji može programe zasebno učitati korištenjem modula, za ARM trenutno nije pronađeno prikladno rješenje.

Drugi je razlog u različitom konceptu zaštite preko segmenata spremnika. Kod arhitekture i386 korištenjem zaštite preko segmenata (i pripadnih registara) ostvareno je odvajanje adresnog prostora na način da se u pojedinim procesima adrese kreću od 0 pa do neke najveće (logičke/relativne adrese), a koje su mapirane na zadani segment. Kod ARM-a adrese i dalje ostaju u fizičkom obliku jedino se za segmente definiraju prava pristupa. Iako bi se funkcionalno moglo i na ovaj način dobiti ista zaštita, ovdje nije prisutna pretvorba adresa iz logične u fizičku i obratno što je bio jedan od glavnih edukacijskih ciljeva uvođenja zaštite preko segmenata u jezgru.

Ostvarenje procesa za platformu ARM odgođeno je za buduće inačice Benua koje će imati i straničenje.

Dodatak B - Upute za korištenje razvojnih alata

B.1. Razvojni alati

U odjeljku 3.2. navedeni su neophodni razvojni alati (kada je razvojno računalo na Linuxu): *GCC*, *ld* (*Binutils*), *Make*, *QEMU*. Osim navedenih koji se koriste za izgradnju i pokretanje sustava, za rad mogu biti potrebni/poželjni i neki drugi. Primjerice, detaljno ispitivanje rada sustava moguće je kada se koristi alat GDB kako je pokazano u odjeljku 5.5. Razlika među inkrementima i fazama može se vidjeti raznima alatima, od tekstualnih, kao što je *diff*, do grafičkih, kao što je *Meld*. Navedeni se alati jednostavno dodaju (*instaliraju*) u okruženju Linuxa. Ovisno o inačici sustava koriste se razni alati, bilo iz grafičkog okruženja ili naredbama u ljuski. Primjerice za sustave koji koriste APT (engl. *advanced packaging tool*), dodavanje svih navedenih programa obavilo bi se naredbama:

```
# Instalacija dodatnih potrebnih/poželjnih programa
# (ako već i jesu instalirani, neće javiti grešku ili će ih pokušati
# nadograditi na zadnju inačicu)

sudo apt-get update
sudo apt-get install gcc libc6-dev binutils make
sudo apt-get install qemu qemu-system mkisofs gdb git-core

# "man" stranice
sudo apt-get install manpages manpages-dev manpages-posix
sudo apt-get install manpages-posix-dev glibc-doc

# opcionalno (meld - usporedba datoteka i direktorija; kate - editor)
sudo apt-get install meld kate

# opcionalno, za primjere iz 2. poglavlja (Linux, Busybox, U-Boot)
sudo apt-get install libncurses5-dev
sudo apt-get install gcc-arm-linux-gnueabi libc6-dev-armel-cross u-boot-tools

# izgradnja 32-bitovnih sustava na 64-bitovnim arhitekturama
sudo apt-get install gcc-multilib

# GCC ARM EABI
sudo apt-get install gcc-arm-none-eabi
# ili Sourcery CodeBench Lite Edition for ARM EABI
/bin/sh arm-2013.11-24-arm-none-eabi.bin -console
```

Upute za korištenje alata nalaze se na web stranicama alata. U ovom su dodatku u nastavku samo istaknute neke posebnosti alata kada ih se koristi zajedno s Benu sustavom. Alati *Make* i *GCC* su već opisani u poglavljima 3., 4. i 5.

B.2. Postavke virtualizacijskih alata

Izgrađeni sustav potrebno je negdje isprobati. Iako bi se on mogao pokretati na pravom računalu, takav razvojni proces bi bio vrlo spor. Naime, izgrađeni sustav bi trebalo pohraniti na neki medij te onda ugasiti računalo i pokrenuti ga s tog medija. S obzirom na to da je izgrađeni sustav vrlo jednostavan i nezahtjevan, bolje ga je pokrenuti u virtualnom okruženju (na simuliranom računalu). Razvoj emulatora je toliko napredovao da se i samo razvojno računalo također može pokretati u emulatoru. Stoga nekoliko osnovnih crtica za postavke.

Ako se i samo razvojno računalo simulira alatima kao što su VMware Player [VMware] i *VirtualBox* [VirtualBox], onda je potrebno pripremiti takve sustave. Najjednostavnije je pronaći

gotovu sliku jednog takvog sustava na Internetu i samo ga prilagoditi – dodati potrebne programe. Suvremeni razvojni sustav temeljen na Linuxu može ugodno raditi u grafičkom okruženju ako mu se u emulatoru dodijeli oko 512 MB spremnika ili više (neki sustavi će raditi i s manje, od 256 MB na dalje). Zahtjevi prema diskovnom prostoru ovise o inačici sustava koji se pokreće, a kreću se od nekoliko GB do 10 GB prostora diska. Radi jednostavnije razmjene podataka između početnog operacijskog sustava (engl. *host OS*) i virtualnog računala pokretanog u emulatoru (engl. *guest OS*) poželjno je po instalaciji i pokretanju sustava dodatno prilagoditi virtualno računalo korištenjem dodataka emulatora (engl. *VMware Tools* za VMware Player, *Guest Additions* za *VirtualBox*). Oni će osim jednostavnije razmjene podataka, omogućiti i veću učinkovitost, bolji prikaz i podršku za sklopovlje. Način postavljanja dodataka ovisi o emulatoru: za VMware Player se započinje s dodavanjem CD uređaja (preko kratice *Install VMware Tools* ili slično). Najjednostavniji prijenos podataka je preko dijeljenih direktorija (engl. *shared folders*). Drugi načini uključuju prijenos preko mreže, primjerice pristup dijeljenim direktorijima početnog računala iz virtualnog ili korištenjem poslužitelja podataka (*FTP*, *Git*, *svn* i slični) ili i običan *copy/paste* ako alat to dozvoljava.

Izgrađeni sustav (nakon pokretanja `make`) jest slika sustava u formatu ELF (`.elf` datoteka). Takva se slika može koristiti u simulaciji korištenjem emulatora kao što je to QEMU (i vjerojatno drugih). Slika sustava se također može ugraditi u sliku CD-a koja se onda može pokretati i u mnoštvo drugih emulatora, ali i na pravim sustavima (samo za arhitekturu x86).

Ime programa emulatora QEMU za arhitekturu x86 se do sada mijenjao: počevši s `qemu` do `qemu-system-i386`. Ako se pri pokretanju emulatora s `make qemu` emulator ne pokrene, treba potražiti pravo ime emulatora QEMU (sam *Make* pokušava s `qemu`, `qemu-system-i386` te `qemu-system-x86_64`). Ako je neka druga greška (nedostaje neka zastavica) potrebno je pogledati upute emulatora. emulator QEMU za ARM pokreće se programom `qemu-system-arm`.

B.3. Git – sustav upravljanja izvornim kodovima

Git [Git] je raspodijeljeni sustav za upravljanje revizijama izvornog koda. Razvijen od strane Linusa Torwaldsa za razvoj Linuxa (2005.). Osnovna ideja pri njegovu stvaranju je bila da sustav bude brz i bez nepotrebne komunikacije s poslužiteljem. Zato je svaka kopija repozitorija potpuna, tj. sadrži cijelu povijest razvoja – nije potrebna komunikacija s poslužiteljem za operacije. Poslužitelj služi kao središnji repozitorij na koji se promjene postavljaju i od kuda razni korisnici mogu dohvatiti kod. Git je vrlo moćan te stoga i vrlo složen sustav. Zato se u ovim kratkim uputama spominju samo neke jednostavnije operacije.

Slika B.1. prikazuje sustav središnjeg i lokalnog repozitorija, korištene nazive te osnovne operacije.

Slika B.1. Središnji i lokalni git repozitoriji

Primjerice, prvi dohvat koda s [Benu] (samo za čitanje):

```
$ cd
$ git clone git://git.code.sf.net/p/benu/code oszur
```

Naknadno ažuriranje lokalne kopije s primjenama koje su drugi postavili na poslužitelj:

```
$ cd ~/oszur
```

```
$ git pull origin master
```

Navedenom naredbom ažurira se “master” grana – osnovna/početna grana. Što je to grana i vrlo kratko o radu s granama nalazi se na kraju ovih uputa.

B.3.1. Pojmovi koji se koriste u opisu rada gita

Repozitorij na poslužitelju sadrži prihvaćene promjene.

Lokalni repozitorij (npr. direktorij `oszur`) sadrži i prihvaćene promjene i radnu inačicu.

Izvorni termin za prihvaćene promjene jest index (smješten u `.git` poddirektoriju).

Izvorni termin za radnu inačicu jest working tree (sve ostalo osim `.git` poddirektorija).

Svaka izmjena koja se iz radne inačice zapiše u repozitorij naziva se promjena – izvorni termin jest commit.

B.3.2. Primjer korištenja osnovnih naredbi

Primjer je prikazan u nekoliko koraka.

1. Dohvat koda iz repozitorija na poslužitelju te početne postavke

```
$ git clone adresa ime-dir
```

Dohvat ide prema gornjim uputama (uz zamjenu prave adrese na `adresa`) ili drukčije ako se otvara repozitorij nad kojim se mogu raditi i promjene. Dok repozitorij nije mijenjan, tj. dok je isti kao i u repozitoriju on je čist (engl. *clean*).

```
$ cd ime-dir
```

Postavljanje svog imena za projekt (ako već nije globalno postavljeno) obavlja se s `git config`:

```
$ git config user.name "Ime Prezime"
$ git config user.email "email@adresa.xx"
```

2. Promjena u radnoj inačici

Neka je dohvaćen repozitorij i spremljen lokalno te napravljena promjena u dvije datoteke, prema:

```
$ git clone git://git.code.sf.net/p/benu/code oszur
Cloning into 'oszur'...
remote: Counting objects: 1656, done.
remote: Compressing objects: 100% (698/698), done.
remote: Total 1656 (delta 926), reused 1570 (delta 843)
Receiving objects: 100% (1656/1656), 443.29 KiB | 242.00 KiB/s, done.
Resolving deltas: 100% (926/926), done.
Checking connectivity... done.
$ cd oszur/
$ vi README
$ vi Coding_style
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   Coding_style
        modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

Naredbom `git status` ispisuje se stanje radnog direktorija u kojem se vide datoteke s promjenama. Same promjene u kodu mogu se vidjeti naredbom `git diff`

```
diff --git a/Coding_style b/Coding_style
index 87d9154..1e374db 100644
--- a/Coding_style
+++ b/Coding_style
@@ -79,3 +79,4 @@ example2:

 In example2 indentation in "if" is skipped so that code would not go to far
 right and that function calls would not be extremely broken across lines.
+
diff --git a/README b/README
index 9c4c9b2..bd838e8 100644
--- a/README
+++ b/README
@@ -10,16 +10,20 @@ System is "work in progress", improved/extended with (almost)
every year.

 Name "Benu", besides acronym, represents an Egyptian bird ...

-Compilers used:
+Tools required for building:
 * for i386 - GNU gcc, GNU ld (GNU Binutils)
-* for arm - Sourcery CodeBench Lite Edition for ARM EABI
+
+* for ARM - package for Ubuntu: gcc-arm-none-eabi
+          - if using Sourcery CodeBench Lite Edition for ARM EABI:
+            add -lm to LDFLAGS (Chapter_*/*/arch/arm/config.ini)
+
[...]
```

Promjene u pojedinoj datoteci mogu se vidjeti dodavanjem imena te datoteke na kraj naredbe. Također je moguće uspoređivati razlike među granama, među do sada prihvaćenim promjenama i slično.

Nisu sve promjene uvijek na bolje. Ako nismo zadovoljni s nekom promjenom koja još nije u repozitoriju, nju jednostavno odbacimo tako da iz repozitorija dohvatimo zadnju potvrđenu inačicu s `git checkout ime_datoteke`.

3. Dodavanje promjene u repozitorij

Kada neku promjenu u radnoj inačici želimo uključiti u iduću promjenu u repozitoriju, onda ju prvo moramo označiti za prihvaćanje s `git add ime_datoteke` te potom dodati u prihvaćene promjene s `git commit -m "opis promjene"`.

```
$ git add Coding_style README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Coding_style
        modified:   README

$ git commit -m "sitni komentari"
[master 3d6e048] sitni komentari
```

```
 2 files changed, 16 insertions(+), 10 deletions(-)
```

Ako ima više datoteka s promjenama i sve promjene želimo uključiti u repozitorij, umjesto zadavanja imena svih datoteka može se koristiti zastavica `-A`, tj. naredba `git add -A`.

U slučaju potrebe uvijek se možemo vratiti i na neku prijašnju inačicu sustava i nju pregledavati. Primjerice s `git log` možemo dohvatiti identifikatore zadnjih nekoliko promjena te s `git checkout ID-PROMJENE` dovatiti i stanje sustava u tom trenutku (s tom promjenom).

```
$ git log
commit 3d6e048bf47e7e85aa16a68b784e8017fa370dd5
Author: Leonardo Jelenkovic <leonardo.jelenkovic@fer.hr>
Date:   Thu Oct 2 14:53:28 2014 +0200

    sitni komentari

commit 7961e3d53b91f4809c56b6cadd9e809602d7dacb
Author: Leonardo Jelenkovic <leonardo.jelenkovic@gmail.com>
Date:   Wed Mar 5 13:58:52 2014 +0100

    added .gitignore
[...]
$ git checkout 7961e3d53b91f4809c56b6cadd9e809602d7dacb
Note: checking out '7961e3d53b91f4809c56b6cadd9e809602d7dacb'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 7961e3d... added .gitignore
```

Ako bismo željeli nju mijenjati, najbolje bi bilo napraviti novu granu u toj inačici i dalje raditi u toj grani. Ako smo baš sigurni da želimo odbaciti neke promjene, onda se možemo "trajno vratiti" u to prošlo stanje. Naredbe za to zahtjevaju malo više obrazloženja te nisu ovdje navedene.

4. Postavljanje promjena u središnji repozitorij

Postavljanje je moguće samo ako korisnik ima odgovarajuće dozvole. Uz pretpostavku da trenutni korisnik ima takve dozvole, postupak postavljanja bi izgledao slično navedenome u nastavku.

```
$ git push origin master
```

Promjene (u osnovnoj grani) su sada vidljive i u središnjem repozitoriju.

5. Osnovni rad s granama

Prilikom ispitivanja nekih mogućih promjena poželjno je takve promjene raditi u zasebnim granama, a ne u osnovnoj grani – master.

Nova se grana stvara s `git branch ime_grane`. Odabir te grane za rad u njoj (da radna inačica – lokalne datoteke pripadaju toj grani) postiže se s `git branch ime_grane`. U trenutku stvaranja grane (prema gornjoj naredbi), trenutno stanje se kopira u tu granu – nova grana je kopija trenutnog stanja (s ovako zadanom naredbom). Međutim, svaka promjena u toj grani vidljiva je samo u toj grani. Također, ako se radi promjena u drugim granama promjene su vidljive samo u tim granama.

Primjerice, ako se želi da svaka laboratorijska vježba bude u svojoj grani, onda bismo mogli koristiti sljedeće naredbe.

```
$ git branch lab1
$ git branch lab2
$ git branch lab3
$ git checkout lab1
Switched to branch 'lab1'
$
[raditi potrebne promjene za lab1 te napraviti "commit"]
$ git push origin lab1
[...]
$
$ git checkout lab2
Switched to branch 'lab2'
$
[raditi potrebne promjene za lab2 te napraviti "commit"]
$ git push origin lab2
[...]
$
$ git checkout lab3
Switched to branch 'lab3'
$
[raditi potrebne promjene za lab3 te napraviti "commit"]
$ git push origin lab3
[...]
```

Svaka grana kreće od istog početnog stanja. Prebacivanje među granama postiže se naredbom `git checkout ime_grane`. Poželjno je prije prebacivanja u drugu granu sve lokalne promjene koje nisu prihvaćene prvo prihvatiti u toj grani (napraviti "commit").

Postavljanje grana u središnji repozitorij moguće je napraviti na više načina. Jedan je prikazan u gornjem primjeru (`git push origin ime-grane`).

Spajanje trenutne grane s nekom drugom može se napraviti naredbom:
`git merge neka_grana`
koja će trenutno aktivnu granu spojiti s granom `neka_grana`. Spajanje se neće dogoditi ako to nije moguće napraviti, tj. ako postoji dvojba kako spojiti neke datoteke, primjerice ako na istom mjestu u datotekama ovih grana piše `a=1` u prvoj te `a=2` u drugoj.

Dodatak C - Izdvojene mogućnosti C-a

Programski jezik C (kao mnogi drugi) ima dodatnu podršku za upravljanje pri stvaranju kôda, odvajanje modula, optimizaranja, makroa i slično. U ovom dodatku su ukratko prikazane neke mogućnosti koje C nudi.

C.1. Proširene deklaracije varijabli i funkcija

Ključne riječi `static` i `extern` imaju ponešto drukčija značenja ovisno na što se odnose: na varijablu ili na funkciju.

C.1.1. Varijable

Varijabla se osim definicije tipa podataka koji varijabla sadrži (primjerice `int var;`) može dodatno definirati ključnim riječima `static`, `volatile` i `extern`, proširujući deklaraciju (primjerice `volatile static int var`).

Oznaka `static`

Globalna varijabla deklarirana u nekoj datoteci (izvan svih funkcija) i dodatno označena sa `static` vidljiva je samo unutar kôda te datoteke. Takva je varijabla obično dio podatkovne strukture modula koji se izgrađuje od navedene datoteke. Ona se izravno ne može koristiti izvan nje, već samo preko funkcija ostvarenih u toj istoj datoteci. Na neki način korištenje oznake `static` omogućava skrivanje internih struktura podataka, kao što to rade privatne varijable razreda kod objektno orijentiranih programskih jezika. Različiti moduli, tj. datoteke mogu imati istoimene lokalne varijable ako su označene sa `static` – svaka datoteka koristi svoju. Pri prevođenju, varijabla označena sa `static` neće biti postavljena u globalnu tablicu simbola (i time izazvati grešku ako su različite varijable istog imena deklarirane u više datoteka). Globalna varijabla se pri prevođenju smješta u `.data` ili `.bss` odjeljak, ovisno o tome je li inicijalizirana pri deklaraciji ili nije. Benu intenzivno koristi skrivanje varijabli podsustava (sa `static`). Gotovo svaki podsustav, tj. `.c` datoteka u jezgri ima vlastitu strukturu podataka pohranjenu lokalno na navedeni način.

Lokalna varijabla deklarirana unutar neke funkcije može također biti označena kao `static` . Takva oznaka mijenja način (mjesto) pohrane takve varijable. Kada lokalna varijabla nije `static` ("normalna lokalna varijabla") ona je privremena, stvara se na stogu dretve koja poziva funkciju gdje je varijabla deklarirana. Po završetku funkcije, varijabla nestaje (zajedno s tim dijelom stoga pri povratku iz funkcije). Kada se lokalna varijabla označi i sa `static` onda ona postaje slična globalnoj varijabli označenoj sa `static` jer više nije privremena i nije spremljena na stog već stalno zauzima dio spremnika (pohranjuje se u `.data` ili `.bss` ovisno je li pri deklaraciji i inicijalizirana). Razlika lokalne `static` varijable u odnosu na globalnu jest u njenoj vidljivosti: lokalna je vidljiva samo unutar funkcije u kojoj je deklarirana. Benu ne koristi lokalne `static` varijable jer su sve strukture podataka podsustava lokalne za podsustav, tj. datoteku (globalne `static`).

Oznaka `extern`

Oznaka `extern` koristi se kada se iz jedne datoteke želi koristiti globalna varijabla (koja nije označena i sa `static`) iz neke druge datoteke. Da bi prevoditelj znao da takva varijabla postoji i u drugoj datoteci (u kojoj varijabla nije definirana) treba naznačiti da postoji korištenjem ključne riječi `extern`.

Primjerice, ako je u datoteci `prva.c` deklarirana globalna varijabla `int prva;` ona se može koristiti i iz druge ako se prethodno navede `extern int prva;`.

Varijabla može biti deklarirana i u asemblerskoj datototeci (`.S`) ne samo u `.c` datoteci i obratno, može se sličnom naredbom naznačiti i u asemblerskoj datoteci (s `.extern`) ako je prethodno deklarirana u `.c` datoteci.

Oznaka `volatile`

Oznaka `volatile` uz varijablu kaže da je varijabla "nestabilna", tj. može se promijeniti nenadano te se zbog toga ne smije pohranjivati na dulje periode u registar procesora već pri svakom korištenju nanovo ispitati (pročitati) njenu vrijednost (i pohraniti pri promjeni). Problem s korištenjem varijabli nastaje kada se one paraleno koriste iz više dretvi ili iz dretve i iz obrade asinkronih događaja kao što su prekidi i signali. Naime, pri optimiranju kôda prevoditelj nastoji varijable što duže zadržati u registrima procesora jer je tada pristup do njih najbrži. Međutim, u paralelnom korištenju takve varijable, promjene koje nastaju na jednom mjestu nisu odmah vidljive u spremniku, tj. na drugom mjestu. To može uzrokovati i grešku.

Načelni pristup pri rješavanju gornjeg problema uključuje korištenje sinkronizacijskih mehanizama koji će onemogućiti istovremeni pristup dijeljenim sredstvima. Funkcije sinkronizacije uključuju implicitno osvježavanje varijabli (ažuriranje vrijednosti u/iz spremnika) pri zaključavanju i pri otključavanju.

Problem sa sinkronizacijom jest da se ona ne može svugdje primijeniti. Primjerice, pri obradi prekida nije moguće pozvati sinkronizacijsku funkciju koja ima mogućnost zaustavljanja dretve s obzirom na to da se obrada prekida obavlja sa zabranjenim prekidanjem te bi takvo zaustavljanje zaustavilo cijeli sustav. Tek u proširenom obliku obrade prekida, kad se on obrađuje u novim dretvama to bi bilo moguće.

Ključna riječ `volatile` uz neku varijablu nalaže prevoditelju da takvu varijablu ne sprema u registre procesora pri optimizaciji, odnosno da se ona što je moguće kraće zadržava u procesoru – svaki put kada je potrebna dohvaća se iz spremnika te se pri svakoj promjeni odmah sprema njena nova vrijednost u spremnik. Na taj način se postiže veća ažurnost sustava i lakše je osigurati konzistentnost podataka (ali na uštrb brzine).

Druga (slična) primjena ključne riječi `volatile` može biti pri korištenju neke izlazne naprave preko posebnih spremničkih adresa. Naime, ako se samo piše na takve adrese, a ne i čita s njih, prevoditelj može pri optimiranju doći do zaključka da taj dio kôda ne radi ništa korisno te ga stoga maknuti. Takav se primjer pojavio pri korištenju zaslona u načinu izravnog upisivanja u spremnik. Zato je varijabla preko koje se pristupa tom dijelu spremnika (`video` u `arch/i386/drivers/vga_text.c`) označena i s `volatile`:

Isječak kôda C.1. Chapter_08_Processes/06_Processes/arch/i386/drivers/vga_text.c

```
19 /*! starting address of video memory */
20 volatile static unsigned char *video = (void *) VIDEO;
```

Oznaka `restrict`

Oznaka `restrict` [restrict] prisutna je tek od *C99*[1] norme za programski jezik C, a uvedena je s namjerom povećanja mogućnosti optimiranja. Ideja jest da se oznaka `restrict` koristi samo za dodatno definiranje kazaljki, odnosno namjeru njena korištenja. Namjera kaže da će se tako označena kazaljka jedina koristiti za pristup podacima na koje ona pokazuje. Ako prevoditelj ima tu informaciju može ostvariti bolje optimiranje.

Primjerice, ako je zadana funkcija:

[1] *C99* je skraćeno ime za normu ISO/IEC 9899:1999, prethodna norma za C. Nova norma je ISO/IEC 9899:2011, skraćeno *C11*.

```
void zbroji ( int *A, int *B, int *x )
{
    *A += *x;
    *B += *x;
}
```

prevoditelj ne smije pretpostaviti da su vrijednosti svih parametara različiti – primjerice `x` bi mogao biti jednak parametru `A`. Međutim, ako bi bili različiti, on bi mogao samo jednom učitati vrijednost s adrese `x` i time izbjeći jednu instrukciju. Kada su kazaljke na veće strukture podataka uštede mogu biti i veće.

U Benu se nije koristila takva napomena prevoditelju, ali se `restrict` pojavljuje u opisima POSIX sučelja na [POSIX].

C.1.2. Funkcije

Slično kao i kod varijabli ključne riječi mogu dodatno specificirati funkcije, njihovu vidljivost, način prevođenja i optimiranja.

Oznake `static` i `extern`

Oznaka `static` uz funkciju ograničava njeno korištenje (vidljivost) na datoteku u kojoj je ona definirana.

Suprotno od `static`, funkcija koja je deklarirana u nekoj drugoj datoteci može se koristiti i iz drugih datoteka – dovoljno je prevoditelju reći da takva funkcija postoji navođenjem prototipa te funckije (zaglavlja bez tijela funkcije) sa ili bez dodatne oznake `extern` (prevoditelj će pri prevođenju "vjerovati" da takva funkcija postoji, dok će ju pri povezivanju i potražiti te javiti grešku ako ju ne nađe).

Oznake `inline` i `static inline`

Oznakom `inline` želi se pomoći pri optimiranju kôda – optimiranju brzine rada, tako da se kratke funkcije ne pozivaju kao funkcije nego ugrade na mjesto poziva. Ugradnjom se izbjegava prijenos parametara u funkciju preko stoga, ali i povećavaju mogućnosti optimiranja s obzirom na to da je kôd funkcije sad ugrađen zajedno s ostalim kôdom. Oznaka `inline` je samo preporuka prevoditelju – on ju ne mora poštivati (ako procijeni da se ugradnja funkcije ne isplati ili je nekim drugim zastavicama to onemogućeno, primjerice prevođenjem za ispitivanje rada sustava, tzv. *debugiranje*).

Kraće funkcije se često ostvaruju u `.h` datotekama umjesto u `.c` datotekama iz razloga što ih se želi ugraditi na mjesto poziva iz različitih datoteka (slično makroima). Takve funkcije moraju biti označene sa `static inline`. Oznaka `inline` je podrazumijevana – zato je funkcija i postavljena u `.h` datoteku. Oznaka `static` je potrebna zato što prevoditelj ne mora poslušati sve `inline` oznake, tj. neka od funkcija ne mora biti ugrađena već pozvana. To stvara problem ako se iz više `.c` datoteka uključi ista funkcija (iz iste `.h` datoteke), koja, iako označena s `inline` nije ugrađena već ostvarena (prevedena) kao obična funkcija. Pri povezivanju tih datoteka u jedan program i pri stvaranju tablice simbola došlo bi do greške: isti simbol – ime takve funkcije, pojavilo bi se više puta. Da se to ne dogodi koristi se i oznaka `static` koja će ograničiti vidljivost te funkcije samo na tu `.c` datoteku, tj. njeno ime neće biti u globalnoj tablici simbola.

C.2. Makroi i druge naredbe prevoditelju

Prevođenje izvornog kôda se najčešće obavlja u dva koraka. U prvom koraku – prvom prolasku kroz kôd (engl. *preprocessing*) izračunavaju se konstante i makroi te uvrštavaju u izvorni kôd

koji se tada prosljeđuje u drugi korak prevođenja. Za prvi korak postoje zasebne naredbe koje omogućavaju finije podešavanje načina prevođenja.

C.2.1. Izbjegavanje višestrukog uključivanja zaglavlja

S obzirom na to da jedna struktura podataka može uključivati elemente druge, vrlo često se iz jedne datoteke zaglavlja uključuje i druga i treća i slično. U nekoj datoteci s izvornim kôdom (`.c`) se stoga može dogoditi da se zbog takvih tranzitivnih uključenja jedno zaglavlje uključi više puta. S obzirom na to da takva datoteka može sadržavati definiciju struktura podataka, pri prevođenju se javlja greška zbog višestruke definicije. Jedan od načina rješavanja tog problema jest spriječiti višestruko uključivanje.

Prvi način ("stariji") način sprečavanja višestrukog uključivanja jest korištenjem naredbi `#ifdef`, `#define` i `#endif`.

Primjer C.1. Naredbe `#ifdef`, `#define`, `#endif` i `#pragma once`

Neka se zaglavlje `primjer.h` sastoji od deklaracije strukture `neki_podaci`. Zaštita od višestrukog uključivanja bi za to zaglavlje moglo biti:

```
1  /* početak datoteke */
2
3  #ifndef _PRIMJER_H_
4  #define _PRIMJER_H_
5
6  /* definicije i deklaracije zaglavlja, primjerice: */
7  struct neki_podaci
8  {
9          int    pod1;
10         char   pod2;
11         void  *ptr;
12 };
13
14 #endif /* _PRIMJER_H_ */
```

Ako se pri prevođenju jedne datoteke (`.c`) drugi put učitava isto zaglavlje – `primjer.h`, zbog toga što je već u prvom učitavanju definirana varijabla `_PRIMJER_H_` u drugom se dio unutar `#ifndef` do `#endif` neće učitati.

Drugi način za ostvarenje istog jest koristiti naredbu `#pragma once` koja nalaže da se zadano zaglavlje (sve iza te naredbe) ne učitava više od jednom. To je novija naredba (nije podržana u starijim prevoditeljima).

Primjer C.1. bi s tom naredbom izgledao:

```
1  /* početak datoteke */
2
3  #pragma once
4
5  /* definicije i deklaracije zaglavlja, primjerice: */
6  struct neki_podaci
7  {
8          int    pod1;
9          char   pod2;
10         void  *ptr;
11 };
```

Navedeni pristupi prikazani su i na početku 5. poglavlja.

C.2.2. Međuovisnost struktura u različitim zaglavljima

Problemi sa zaglavljima mogu nastati i ako oni definiraju strukture podataka koje su međusobno ovisne – uključuju elemente one druge strukture.

Primjer C.2. Greška zbog međuovisnosti i moguća rješenja

Neka, za ilustraciju, postoje dva zaglavlja: `time.h` koji definira strukturu `struct ktimer` te `thread.h` koji definira `struct kthread`. Neka u početnom ostvarenju ta zaglavlja izgledaju kao u nastavku.

time.h

```
1  #pragma once
2
3  #include <thread.h>
4
5  struct ktimer
6  {
7          ... /* sve ostalo potrebno za alarm */
8
9          struct kthread *thread;
10 };
```

thread.h

```
1  #pragma once
2
3  #include <time.h>
4
5  struct kthread
6  {
7          ... /* sve ostalo potrebno za opisnik */
8
9          struct ktimer *alarm;
10 };
```

Pri prevođenju će se pojaviti greška jer za potpunu deklaraciju nedostaje onaj drugi tip. Primjerice, ako datoteka s izvornim kôdom uključuje `thread.h` onda će se pri njegovu uključivanju uključiti i `time.h` (zbog `#include <time.h>` na njegovu početku) te pri prevođenju javiti grešku u elementu strukture `struct ktimer`, tj. javiti će da `struct kthread`.

Jedan od načina rješavanja tog problema jest deklaracijom da takva struktura postoji. Sama definicija strukture mora biti negdje iza u datoteci (i bit će kad se oba zaglavlja uključe).

Na prethodnom primjeru bi trebalo dodati `struct kthread;` ispred početa deklaracije `struct ktimer` i obratno, `struct ktimer;` ispred `struct kthread`.

time.h

```
1  #pragma once
2
3  #include <thread.h>
4
5  struct kthread;
6
7  struct ktimer
8  {
9          ... /* sve ostalo potrebno za alarm */
10
11         struct kthread *thread;
```

```
};
```

thread.h

```
#pragma once

#include <time.h>

struct ktimer;

struct kthread
{
        ... /* sve ostalo potrebno za opisnik */

        struct ktimer *alarm;
};
```

Ako su elementi strukture samo kazaljke na drugu strukturu, kao u gornjem primjeru te ako se izravno ne koriste elementi te druge strukture, već isključivo preko funkcija, tada se gornji primjer može pojednostaviti. Prvo pojednostavljenje je da se umjesto navođenja tipa kazaljke (`struct kthread *thread;` i `struct ktimer *alarm;`) koristi opća kazaljka: `void *thread` i `void *alarm`. Drugi je način to isto, ali davanjem istoj strukturi dva značenja: jedno u jednoj datoteci, a drugo u ostalima. Navedeni je pristup korišten u Benu, gdje se interne strukture pojedinog podsustava skrivaju od drugih, odnosno s obzirom na to da se izvan koriste samo kazaljke one su tipa `void`. Primjeri takva skrivanja vidljivi su u sloju jezgre. U nastavku je prikazan dio datoteke `kernel/time.h`.

kernel/time.h

```
#ifndef _K_TIME_C_
typedef void ktimer_t;
#else
struct _ktimer_t_; typedef struct _ktimer_t_ ktimer_t;
#endif /* _K_TIME_C_ */
```

Za sve datoteke, tip podataka `ktimer_t *` je obična kazaljka `void *`, osim za `kernel/time.c` koji definira makro `_K_TIME_C_` za koji je to prava struktura.

C.2.3. Dodatno postavljanje svojstava podataka i kôda

Naredba `__attribute__` omogućava postavljanje dodatnih svojstava podataka i kôda. Dva koja su ovdje objašnjena su `packed` i `section`.

Atribut `packed`

Struktura deklarirana sa `struct` sadrži elemente ista ili različita tipa podataka. Njena osnovna namjena je da drži zadane elemente zajedno. Stoga kada se u programu pristupa pojedinom elementu, prevoditelj točno zna (izračuna) gdje je taj element smješten gledajući od početka strukture. Kada je pojedini element manji od riječi procesora (uobičajene veličine poravnavanja podataka) prevoditelj će za podatak ipak zauzeti ipak cijelu riječ radi jednostavnjeg i bržeg pristupa u toj arhitekturi. Primjerice, ako struktura sadrži podatke:

```
struct svasta
{
        int broj1;
        char znak;
        double broj2;
        char polje[10];
};
```

u 32-bitovnoj arhitekturi prevoditelj će napraviti sljedeću strukturu:

- `broj1` će zauzeti 32 bita = 4 okteta
- `znak` će zauzeti idućih 8 bita = 1 oktet
- iduća tri okteta se neće koristiti radi poravnanja
- `broj2` će zauzeti 64 bita = 8 okteta
- `polje` će zauzeti 10 * 8 bita = 10 okteta
- iduća dva okteta se neće koristiti radi poravnanja.

Razlog takvog poravnavanja (4 + 4 + 8 + 12) jest u bržem pristupu podataka kada su oni poravnati. Kada to ne bi bilo tako, ako bi podaci bili kompaktno spremljeni jedan do drugog, moguće je da bi za dohvat jednog podatka trebalo dva puta do spremnika, jer procesor možda može čitati samo s adresa poravnatih po veličini riječi (ili višekratniku, kao što je to potrebno za tip `double`).

Ponekad takvo ponašanje prevoditelja treba spriječiti. Primjerice, ako struktura opisuje poruku koja se razmijenjuje među različitim računalima, ta poruka mora biti jednoznačno prepoznata na obje strane, bez obzira o prevoditeljima koji su korišteni za programe na svakoj strani i bez obzira na procesore na pojedinim stranama. Primeri takvih poruka su poruke koje se razmijenjuju preko Inteneta – takozvani paketi. Struktura poruke (struktura korisnog dijela, bez zaglavlja koje koriste prijenosti protokoli) može biti definirana u programskom jeziku, primjerice C-u sa `struct`. Da bi poruka bila sažeta i jednako interpretirana treba prevoditeljima dati posebne naredbe, koristeći naredbu `attribute` + `packed`:

```
1  struct svasta
2  {
3          int broj1;
4          char znak;
5          double broj2;
6          char polje[10];
7
8  } __attribute__((__packed__));
```

Struktura će sada biti sažeta, bez dodavanja praznih mjesta (4 + 1 + 8 + 10).

Pri korištenju poruka često se želi definirati tip poruke koja ima proizvoljnu duljinu, odnosno koja sadrži jedan element proizvoljne duljine. Primjerice, neka se poruka koju razmijenjuju dva računala sastoji od tipa poruke (`int` – 4 okteta), duljine (`size_t` – neka je također 4 okteta) te samih podataka proizvoljne duljine (zapravo duljine koja odgovara prethodnom parametru). Korištenjem C-a može se definirati sljedeća struktura:

```
1  struct poruka
2  {
3          int tip;
4          size_t velicina;
5          char poruka[1];
6
7  } __attribute__ ( ( __packed__ ) );
```

Iako je zadnji element – `poruka` definiran kao polje od jednog elementa, on može poslužiti za rad s proizvoljno dugom porukom, naravno pod uvjetom da je ona tamo smještena, kao u sljedećem primjeru.

```
1  int posalji_poruku ( int tip, size_t velicina, char *poruka )
2  {
3          struct poruka *p = malloc ( sizeof(struct poruka) +
4                                      velicina * sizeof (char) - 1 );
5
6          p->tip = tip;
```

```
7         p->velicina = velicina;
8         memcpy ( &p->poruka[0], poruka, velicina );
9
10        posalji_drugom_cvoru ( p );
11 }
```

Atribut `section`

Kod ugrađenih sustava može biti potrebno određeni dio kôda (funkcije) ili dio podataka smjestiti u točno određen dio spremnika. Kao što je prethodno već i prikazano (5.2. i D.) tome prvenstveno služi skripta za povezivanje. Ponekad je potrebno još detaljnije uređivanja smještaja podataka neke datoteke. Primjerice, može biti potrebno da neka struktura podataka neke datoteke bude na jednom mjestu (jednom dijelu spremnika) a druga u drugom (slično je i za instrukcije). Tada je moguće posebnim naredbama (`attribute` + `section`) odrediti u koji će se odjeljak prevesti zadana struktura podataka (ili funkcija), a da bi se kasnije pri povezivanju to moglo tamo i postaviti.

Pri izgradnji programa kao odvojenih modula (kasnije procesa) bilo je potrebno na sam početak modula postaviti zaglavlje koje opisuje taj modul. U prikazanom rješenju (od faze `Chapter_08_Processes/03_Programs_as_modules`) koristi se struktura `proginfo_t` koja se pri povezivanju postavlja na početak (prva i jedina struktura podataka definirana u `api/prog_info.c`). Ako bi u tu datoteku bilo potrebno staviti i druge strukture podataka koje ne trebaju (ili ne smiju) ići na početak modula, onda bismo tu strukturu (`prog_info_t pi`) mogli staviti u posebni odjeljak i samo taj odjeljak staviti na početak modula. Varijablu `pi` bi trebalo dodatno označiti, primjerice:

```
1 prog_info_t pi __attribute__ ((section (".pocetak_modula"))) = ...
```

Slično se može napraviti i s funkcijama:

```
1 void prog_init ( void *args ) __attribute__ ((section (".inicijalizacija_modula")))
2 {
3         ...
4 }
```

U asembleru se isto postiže naredbom `.section ime_odjeljka` navedenom prije kôda ili podataka.

Ostali atributi koji se mogu postaviti za podatke i funkcije ovise o prevoditelju. Opis mogućih za *GCC* mogu se pronaći u [GCC] (poglavlja *Specifying Attributes of Variables* i *Declaring Attributes of Functions*).

C.2.4. Korištenje makroa za kraće operacije

Neki primjeri korištenja makroa već su prikazani u poglavlju 5.5. Radi potpunosti, ovdje su navedene i druge preporuke u njihovu korištenju.

Makroi se izračunavaju u prvom koraku prevođenja (prije "pravog prevođenja"), tj. ugrađuju se na mjesto poziva. Stoga njihovu oblikovanju treba oprezno pristupiti. Općenita je preporuka da se umjesto složenijih makroa pišu zasebne funkcije. Međutim, to nije uvijek moguće. Stoga u nastavku slijedi nekoliko preporuka za korištenje makroa.

Pisanje makroa u više redova

Makro definiran s `#define` mora biti u jednom retku. Međutim, radi preglednosti može se koristiti znak \ za oznaku nastavka reda u novom redu (kao da novi red nastavlja u istom).

Primjer makroa u nekoliko redova:

```
#define ZBROJI(X,Y,Z,W)                          \
```

```
( slozeni_proracun_u_funkciji_duga_imena1(X) +  \
  slozeni_proracun_u_funkciji_duga_imena2(Y) +  \
  slozeni_proracun_u_funkciji_duga_imena3(Z) +  \
  slozeni_proracun_u_funkciji_duga_imena4(W)    \
)
```

Omeđivanje parametara

Parametre makroa treba staviti u zagrade (osim kada to nema smisla) da se u složenijim izrazima ne bi dobili krivi rezultati.

Primjer lošeg makroa:

```
#define MUL(X,Y)    X * Y
```

Poziv gornjeg makroa preko:

```
    X = MUL ( a + b, c - d );
```

dao bi:

```
    X = a + b * c - d;
```

što vjerojatno nije očekivani rezultat. Ispravan makro bi bio:

```
#define MUL(X,Y)    ( (X) * (Y) )
```

Dodatne zagrade oko umnoška mogu spriječiti krivo povezivanje makroa s operatorima još veća prioriteta s lijeve ili desne strane.

Omatanje složenijeg makroa – `do{...}while(0)`

Složeniji izrazi koji uključuju više operacija, ali pritom ne vraćaju vrijednost, mogu se omotati s `do{naredbe;}while(0)` petljom tako da budu uporabljivi u svim dijelovima koda.

Primjer omatanja:

```
#define TEST(X,Y,Z)     \
do {                    \
    if ( (X) > 0 )      \
        (Z) = 1;        \
    else if ( (Y) > 0 ) \
        (Z) = 2;        \
    else                \
        (Z) = 0;        \
}                       \
while (0)
```

Gornji makro može se pozvati u svim dijelovima programa, primjerice u kodu:

```
    if ( uvjet1 )
        TEST ( a, b, status );
    else
        TEST ( c, d, status );
```

Višestruko korištenje parametara

Ako se isti parametar u tijelu makroa javlja više puta makro treba drukčije napisati ako parametri mogu biti složeniji izrazi koji i sami nešto mijenjaju. Parametre treba pridijeliti lokalnim varijablama te dalje raditi s njima, a ne s parametrima.

Primjerice, makro koji bi izračunavao druge korijene kvadratne jednadžbe (uz pretpostavku da su realni) mogao bi se zapisati prema:

```
#define KVJED(A,B,C,X1,X2)                                                  \
do {                                                                        \
    X1 = ( -(B) - sqrt ( (B) * (B) - 4 * (A) * (C) ) ) / ( 2 * (A) );       \
    X2 = ( -(B) + sqrt ( (B) * (B) - 4 * (A) * (C) ) ) / ( 2 * (A) );       \
}                                                                           \
while (0)
```

Problem korištenja makroa ilustriraju primjeri:

```
    KVJED ( i++, --j, k, x1, x2 );

    KVJED ( a = b + c, b = a + c, c = funkcija ( a, b ), x1, x2 );
```

Iako navedeni primjeri korištenja makroa ne izgledaju suvislo, u raznim problemima se mogu pojaviti slične (ali smislene) situacije.

Ispravan makro za prethodni problem možemo napraviti korištenjem dodatnih varijabli unutar `do while` petlje prema:

```
#define KVJED(A,B,C,X1,X2)                                                  \
do {                                                                        \
    double _a_ = (A), _b_ = (B), _c_ = (C);                                 \
    X1 = ( -_b_ - sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );       \
    X2 = ( -_b_ + sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );       \
}                                                                           \
while (0)
```

Neuobičajena imena lokalnih varijabli je potrebno koristiti iz razloga što u slučaju da parametri imaju ista imena makro ne bi radio ispravno. Primjerice, ako bi se prethodni makro pozvao preko:

```
    KVJED ( _a_, m, n, r, s );
```

poziv bi se u prvom koraku prevođenja preveo u:

```
do {                                                                        \
    double _a_ = (_a_), _b_ = (m), _c_ = (n);                               \
    r = ( -_b_ - sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );        \
    s = ( -_b_ + sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );        \
}                                                                           \
while (0)
```

S obzirom na to da u bloku iza `do` lokalna varijabla `_a_` će se koristiti umjesto parametra ("vanjske varijable" `_a_`) te se toj lokalnoj varijabli neće proslijediti željena vrijednost.

Nadalje, prethodni makro pretpostavlja da su parametri tipa `double`. Općenitiji makro možemo napraviti ako prevoditelj (npr. *gcc*) podržava makro `typeof` koji za `typeof (izraz)` vraća tip podatka koju određuje `izraz`.

```
#define KVJED(A,B,C,X1,X2)                                                  \
do {                                                                        \
    typeof (A) _a_ = (A);                                                   \
    typeof (B) _b_ = (B);                                                   \
    typeof (C) _c_ = (C);                                                   \
    X1 = ( -_b_ - sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );       \
    X2 = ( -_b_ + sqrt ( _b_ * _b_ - 4 * _a_ * _c_ ) ) / ( 2 * _a_ );       \
}                                                                           \
while (0)
```

Korištenje uvjetnog pridruživanja ("skraćeni `if`")

Mnogi se makroi mogu riješiti korištenjem uvjetnog pridruživanja, tj korištenja konstrukta:

```
uvjet ? vrijednost_za_ISTINA : vrijednost_za_NEISTINA
```

Navedeni se uvjeti mogu i kombinirati. Primjerice makro:

```
#define SQR_MIN(A,B,C) \
sqr( ((A)<(B)) ? ( ((A)<(C)) ? (A) : (C) ) : ( ((B)<(C)) ? (B) : (C) ) )
```

će izračunati kvadrat najmanje vrijednosti od A, B i C. Navedeni makro pretpostavlja da su parametri jednostavni izrazi, da sami ne mjenjaju nešto. Ako se to ne može pretpostaviti treba koristiti makro sličan prethodnom `KVJED` (koji traži i parametre za pohranu rezultata).

Vraćanje vrijednosti iz složenih makroa (proširenje koje donosi alat *gcc*)

Prevoditelj *gcc* nudi mogućnost povrata vrijednosti i iz složenih makroa preko konstrukta:

```
#define IME_MAKROA(PARAMETRI)   \
(                               \
    tijelo složenog makroa;     \
    povratna_vrijednost;        \
)
```

Ako se u složenom makrou definiraju i lokalne varijable unutarnji dio potrebno je dodatno omotati u blok korištenjem zagrada `{}`.

Primjerice, poopćenje makroa `SQR_MIN`, uz korištenje svake vrijednosti samo jednom, moglo bi se napraviti prema:

```
#define SQR_MIN(A,B,C)      \
({                          \
    typeof (A) _a_ = (A);   \
    typeof (B) _b_ = (B);   \
    typeof (C) _c_ = (C);   \
    typeof (A) min = _a_;   \
                            \
    if ( min > _b_ )        \
        min = _b_;          \
    if ( min > _c_ )        \
        min = _c_;          \
                            \
    sqr( min );             \
})
```

Povratna vrijednost prethodnog makroa je zadnja linija koda: `sqr(min)`. Međutim, navedeno proširenje sintakse vrijedi (trenutno) samo za *gcc*.

Povezivanje simbola – operator ##

Pomoću operatora `##` moguće je stvoriti novo ime spajajući dijelove imena. Pokažimo to na primjeru.

Pretpostavimo da u sustavu imamo nekoliko objekata nad kojima je ostvarena ista operacija dodavanja u listu. Za svaki objekt funkcija dodavanja je imenovana prema tipu objekta uz sufiks `_append_to_list`. Makro kojim ćemo skratiti pisanje te funkcije u kodu mogao bi izgledati prema:

```
#define OBJ_APPEND ( TYPE, OBJ )    TYPE ## _append_to_list ( OBJ )
```

Pozivi:

```
    OBJ_APPEND ( net, socket );
    OBJ_APPEND ( dev, device );
    OBJ_APPEND ( fs, file );
```

prevest će se u:

```
    net_append_to_list ( socket );
```

```
    dev_append_to_list ( device );
    fs_append_to_list ( file );
```

Pretvaranje u niz znakova – operator #

Pri ostvarenju ispisa u dnevnik događaja često nam je potrebna operacija pretvaranja imena varijable u niz znakova i slično. Tome nam može pomoći operator `#`.

Primjerice, makro:

```
#define LOG(VAR)    printf ( #VAR "=%d\n", VAR )
```

pozivom:

```
    LOG ( varijabla_x );
```

prevest će se u:

```
    printf ( "varijabla_x=%d\n", varijabla_x );
```

Varijabilan broj parametara

Varijabilan broj parametara u deklaraciji označavamo s tri točke (`...`), dok u tijelu makroa s `## __VA_ARGS__`. Operator `##` služi da u slučaju nepostojanja parametara (u dijelu varijabilnih parametara) ne izazove grešku (miče se prethodni zarez).

Makro za ispis u dnevnik mogao bi izgledati prema:

```
#define DEBUG(FORMAT,...)   \
fprintf ( log_file, "[%s:%d] " FORMAT "\n", __FILE__, __LINE__,\
          ##__VA_ARGS__ )
```

Pozivi u datoteci `test.c`:

```
15        DEBUG ( "checkpoint-1" );
16
17        DEBUG ( "%d", var1 );
18
19        DEBUG ( "a=%d, b=%f (s=%s)", a, b, s );
```

preveli bi se u:

```
    fprintf ( log_file, "[%s:%d] checkpoint-1", "test.c", 15 \
              );

    fprintf ( log_file, "[%s:%d] %d", "test.c", 17, \
              var1 );

    fprintf ( log_file, "[%s:%d] a=%d, b=%f (s=%s)", "test.c", 19, \
              a, b, s );
```

Jedno od ograničenja gornjeg makroa jest u korištenju statički definiranog formata za parametar `FORMAT` – ne može se format prenijeti kao varijabla.

C.2.5. Ostali operatori i naredbe

U datoteci s izvornim kôdom u programskom jeziku C moguće je navoditi i asemblerske instrukcije. Ključna riječ/makro `asm` omogućava unošenje strojnih instrukcija. Primjeri korištenja takvog ubačenog asembler (engl. *inline assembly*) nalaze se u odjeljku 5.4. Ubačeni asembler se u fazi optimiranja može i promijeniti (optimirati). Ako zbog optimiranja mogu nastati problemi, može se dodati i ključna riječ `volatile` kojom se nalaže prevoditelju da navedeno mora biti kako je navedeno – da se ne mijenja u postupku optimiranja.

Dodatak D - Primjeri skripte za povezivanje

Priprema programske komponente za ugrađene sustave koji se obično sastoje od barem dva tipa spremnika: ROM i RAM, zahtjeva dobro poznavanje načina izgradnje slike sustava (engl. *image*). Način izgradnje sustava i skripte koje se pritom koriste već su prikazane u poglavljima 5.2., 12.4. i 12.5.2. Ovaj dodatak proširuje prikaze korištenja skripti dodatnim primjerima (preporučuje se prvo pogledati navedena poglavlja prije korištenja ovog).

D.1. Sažetak o povezivanju – osnovni postupci na primjeru

Neka je izvorni kod zadan datotekama `a.c`, `b.c` i `c.c`. Postupak prevođenja prikazan je slikom D.1.

Slika D.1. Postupak prevođenja i pokretanja – osnovni postupci

Prvi korak u izgradnji, prevođenje naredbom `gcc -c x.c`, stvorit će objektne datoteke `a.o`, `b.o` i `c.o`. Sadržaj objektnih datoteka su tablice sa simbolima, instrukcije, konstante i podaci, složeni u odjeljke `.text`, `.rodata`, `.data` i `.bss`. Program *gcc* će napraviti i još neke, ali oni nisu toliko bitni za ova razmatranja te se neće spominjati u nastavku.

Povezivanje (linkanje) predstavlja spajanje tih objektnih datoteka. Ako nema posebnih uputa,

povezivač (linker, *ld*) će istoimene odjeljke iz različitih datoteka spojiti i tako stvoriti izlaznu datoteku. Ako ima posebnih uputa (npr. `ldscript.ld`) onda će linker spajanje napraviti prema uputama. Slika D.2. prikazuje jedan takav primjer.

Slika D.2. Postupak prevođenja i pokretanja – odjeljci u raznim fazama

Pojmovi vezani uz prevođenje i povezivanje:

- ulazni odjeljci: odjeljci iz ulaznih objektnih datoteka
  - npr. u `a.o` nalaze se odjeljci `.text`, `.rodata`, `.data`, `.bss`
- izlazni odjeljci: odjeljci u izlaznoj datoteci
  - nakon povezivanja, nastaju spajanjem ulaznih odjeljaka iz svih korištenih ulaznih objektnih datoteka
  - uobičajeni nazivi: `.text`, `.rodata`, `.data`, `.bss`

  - posebni nazivi: prema uputama u `ldscript.ld` (npr. `.x1`, `.x2`, `.x3`, `.x4`)
- pojmovi vezani uz povezivanje – vezani uz izlazne odjeljke:
  - npr. dio stupaca u ispisu `objdump -h slika.elf`
  - adresa za koju se odjeljak priprema – VMA (engl. *virtual memory address*)
    * broj odmah nakon oznake odjeljka u `ldscript.ld` (npr. `.x1 VMA_ADDR : AT ( LMA_ADDR )`)
    * pri radu (pokretanju programa/sustava) taj odjeljak treba učitati na tu adresu (ili koristiti sklopovlje, tako da izgleda da je na toj adresi, npr. straničenje ili sklop za zbrajanje adresa)
  - adresa na koju odjeljak treba učitati u spremnik – LMA (engl. *logical memory address*)
    * pri pokretanju programa/sustava ili pri spremanju u ROM, odjeljak treba staviti na tu adresu
    * LMA je najčešće jednako VMA, ali ne mora biti (npr. ako se podaci početno zapisuju u ROM prema LMA, ali se pri pokretanju sustava kopiraju u RAM prema VMA da bi se mogli mijenjati)
    * pri pokretanju Benu sustava, QEMU učitava sliku i radi navedeno
  - položaj u izlaznoj datoteci (npr. datoteci `slika.elf`)
    * *file offset*
    * ako je datoteka u formatu ELF onda i nije bitno
    * ako je datoteka u binarnom formatu, onda gornje informacije (VMA, LMA) nisu dostupne – pri učitavanju takve slike sustava ona se doslovce učitava istim redoslijedom navedenim u datoteci

Nakon povezivanja (npr. `ld -o slika.elf a.o b.o c.o -T ldscript.ld`) nastaje izlazna datoteka (`slika.elf`).

D.2. Primjeri sustava

Primjer D.1. Opis sustava

Neka se za neki ugrađeni sustav priprema programska potpora koja se sastoji od operacijskog sustava i programa. Ugrađeni sustav ima 32 MB ROM-a na adresi `0x01000000` te 32 MB RAM-a spremničkog prostora na adresi `0x03000000`. U ROM-u će se u početku nalaziti sve. Pri pokretanju sustava (pri njegovoj inicijalizaciji) neki se podaci kopiraju u RAM, a neki se tamo tek stvaraju. Instrukcije jezgre, konstante koje jezgra koristi te instrukcije svih programa, mogu se cijelo vrijeme nalaziti u ROM-u i iz njega pokretati i koristiti.

Sustav za upravljanje spremnikom koristi straničenje te moguća rascjepkanost spremničkog prostora koji koristi pojedini program (koji u izvođenju postaje proces) neće biti problem (dovoljno je prilagoditi tablicu prevođenja).

Struktura podataka jezgre (opisnici, međuspremnici, ...) mora se kopirati/stvoriti u RAM-u. Spremnički prostor za podatke pojedinih programa, prostor za gomilu te za stogove dretvi, također treba smjestiti u RAM.

S obzirom na to da se za upravljanje spremničkim prostorom procesa (ne i jezgre) koristi straničenje, programe treba pripremiti (prevesti) tako da ostanu u logičkim adresama koje započinju s adresom `0`. Ipak, programi ne koriste prvi MB jer je tu potrebno postaviti (mapirati) neke podatke jezgre koji procesu neće biti dostupni ali su potrebni pri prihvatu

prekida. Drugim riječima, programe treba pripremiti za logičku adresu iza prvog MB, tj. tako da počinju na adresi `0x00100000`.

U logičkoj strukturi programa (logičnom adresnom prostoru) trebaju biti redom:

- prvi MB nedostupan (koristi ga jezgra)
- instrukcije (`.text` odjeljci)
- konstante (`.rodata` odjeljci)
- podaci koji se mogu mijenjati, primjerice globalne varijable (`.data` i `.bss` odjeljci)
- prostor za gomilu
- prostor za stogove dretvi.

Izvorni kodovi jezgre neka se nalaze u direktoriju `kernel`, a programi u `programs`.

Primjer D.2. Jezgra i samo jedan program

Kada bi u sustavu bio samo jedan program, onda bi se jezgra i program mogli povezati u sliku sustava i pomoću samo jedne skripte. U inkrementu `Chapter_08_Processes/04_Programs_as_process`, koji sve programe stavlja u isti proces to nije moguće napraviti jer i jezgra (`kernel/*`) i programi (`programs/*`) koriste funkcije iz pomoćnih biblioteka (`lib/*`), a njih treba posebno pripremiti za jezgru u fizičkim adresama te posebno za programe u logičkim adresama – isti simbol (ime funkcije) ne može imati dvije različite vrijednosti.

Jezgra i jedan program – zajednička skripta

```
OUTPUT_FORMAT ( "elf32-i386" ) /* ili slično za neku drugu arhitekturu
                                * i/ili format */
ENTRY ( kernel_init )       /* početna funkcija za inicijalizaciju */

/* konstante */
ROM_START  = 0x01000000;
RAM_START  = 0x03000000;
PROG_START = 0x00100000;

SECTIONS {
    .kernel_code ROM_START : AT ( ROM_START )
    /* "Pripremi" jezgren kod i konstante tako da budu pripremljeni za
     * početnu adresu ROM_START. Na istu će se adresu i učitati u ROM
     * (AT dio). U ovom slučaju AT dio i nije potreban jer se radi o
     * fizičkim adresama i u prvom slučaju. */
    {
        *kernel/?*( .text* )        /* instrukcije */
        *kernel/?*( .rodata* )      /* konstante */
    }

    kernel_data_in_ROM = .;
    /* . => trenutna "adresa" (location pointer) u ovom položaju ima
     *      vrijednost: ROM_START + SIZEOF(.kernel_code) */

    .kernel_data RAM_START : AT ( kernel_data_in_ROM )
    /* Podaci jezgre su početno u ROM-u (kernel_data_in_ROM), ali se
     * pri pokretanju trebaju programski kopirati u RAM na adresu
     * RAM_START, na početak RAM-a. Zato se i pripremaju za tu
     * adresu, a ne za adresu na kojoj se početno nalaze. */
    {
        *kernel/?*( .data* )
```

```
        *kernel/?*( .bss* )
        . = ALIGN (4096); /* poravnaj kraj na 4 KB */
    }

    program_code_in_ROM = kernel_data_in_ROM + SIZEOF (.kernel_data);

    .program_code PROG_START : AT ( program_code_in_ROM )
    /* Instrukcije i konstante programa se pripremaju za logičke
     * adrese od PROG_START, ali će se učitati odmah iza jezgre u
     * ROM (program_code_in_ROM) i tu ostati. */
    {
        *programs/?*( .text* )  /* instrukcije */
        *programs/?*( .rodata* )    /* konstante */
        . = ALIGN (4096);
    }

    program_data_in_ROM = program_code_in_ROM + SIZEOF (.program_code);
    /* fizička adresa početne kopije podataka u ROM-u */

    program_data_in_RAM = PROG_START + SIZEOF ( .program_code );
    /* logička adresa nakon pokretanja; s obzirom na to da se koristi
     * straničenje, program, njegovi dinamički elementi se mogu
     * učitati bilo gdje u RAM - dovoljno je prilagoditi tablicu
     * prevođenja. */

    .program_data (program_data_in_RAM) : AT ( program_data_in_ROM )
    /* Podaci programa se pripremaju za logičke adrese od
     * 'program_data_in_RAM', ali će se učitati odmah iza
     * '.program_code' u ROM na adresu 'program_data_in_ROM' i od tu
     * pri pokretanju kopirati negdje u RAM. */
    {
        *programs/?*( .data* )  /* inicijalizirani podaci */
        *programs/?*( .bss* )   /* neinicijalizirani podaci */
        . = ALIGN (4096);
    }

    ROM_SIZE = program_data_in_ROM + SIZEOF ( .program_data ) - ROM_START;
    /* ukupna veličina spremničkog prostora koja je potrebna za ROM */

    kernel_heap = RAM_START + SIZEOF ( .kernel_data );
    /* Od tu počinje adresni prostor u RAM-u gdje se mogu spremiti
     * dinamički podaci jezgre (npr. gomila za dinamičko upravljanje
     * spremnikom, dijelovi spremničkog prostora za dinamičke dijelove
     * programa (podatke, gomilu i stog). */

    /* Za stvaranje gomile i rezervaciju mjesta za stog pri pokretanju
     * programa: */

    program_heap = program_data_in_RAM + SIZEOF ( .program_data );
    /* Logička adresa početka spremničkog prostora za gomilu u
     * programu */

    program_stacks = program_heap + PROGRAM_HEAP_SIZE;
    /* Logička adresa početka spremničkog prostora za stogove dretvi u
     * programu. Konstanta PROGRAM_HEAP_SIZE treba biti negdje
     * definirana (primjerice u Makefileu).
     * Drugi pristup može biti u postavjanju mjesta za stogove na kraj
     * (logičkog) adresnog prostora tako da raste prema manjim
     * adresama, što je uobičajen pristup u operacijskim sustavima.
     */
}
```

Pri prevođenju programa izlazna “slika sustava” koju treba pohraniti u ROM bi se sastojala od dijelova prikazanih tablicom D.1.

Tablica D.1. Dijelovi slike sustava

Izlazni odjeljak	Logička adresa	Smještaj u ROM-u	Fizička adresa
`.kernel_code`	`0x01000000`	`0x01000000`	`0x01000000`
`.kernel_data`	`0x03000000`	iza prethodnog odjeljka	`0x03000000` (tu ih treba kopirati pri pokretanju)
`.program_code`	`0x00100000`	iza prethodnog odjeljka	može bilo gdje, ali ostaje u ROM-u, nije potrebno kopirati
`.program_data`	`0x00100000` + veličina od `.program_code`	iza prethodnog odjeljka	početna kopija je u ROM-u; pri pokretanju treba kopirati u RAM

U izvođenju, slika sustava se može prikazati slikom D.3.

Slika D.3. Moguće zauzeće ROM-a i RAM-a tijekom rada

Da bi se postigla slika sustava (prema slici D.3.), pri pokretanju sustava dijelove iz ROM-a treba prekopirati u RAM. Primjerice, za početno kopiranje podataka jezgre u RAM mogla bi se koristiti sljedeća funkcija:

Kopiranje podataka u RAM

```
void copy_kernel_data ()
{
    extern char kernel_data_in_ROM, program_code_in_ROM, RAM_START;
    void *start = &kernel_data_in_ROM;
    void *end = &program_code_in_ROM;
    size_t size = ((size_t) end) - ((size_t) start);

    memcopy ( (void *) &RAM_START, start, size );
}
```

Varijable iz skripte postaju simboli. Njihova vrijednost u skripti odgovara njihovoj adresi. Zato se u prethodnoj funkciji oni označavaju kao varijable, ali se koriste vrijednosti njihovih adresa.

Primjer D.3. Jezgra i više programa

Kada sustav ima više programa od kojih svaki treba biti u zasebnom procesu, tada se može ili koristiti jedna skripta, ali sa zasebnim odjeljcima za svaki program (kao u prethodnom primjeru) ili koristiti jednu skriptu za jezgru te jednu skriptu za sve programe, ali uz zasebno prevođenje svakog programa (uz korištenje te jedne skripte). U nastavku je prikazano zadnje rješenje.

Slika sustava se u ovom primjeru sastoji od slike jezgre i slika svih programa. Za učitavanje takve slike u ROM potreban je poseban alat koji će najprije u ROM učitati jezgru, a nakon nje sve programe (jedan iza drugoga).

Pri pokretanju sustava jezgri treba dati podatke o tome gdje se u ROM-u nalazi svaki program (sa svim svojim dijelovima). Jedan od načina da se to napravi jest da se na početku svakog programa nalazi struktura podataka koja opisuje taj program: njegove segmente i veličinu (slično kao u Benu). Na taj način jezgra može krenuti od adrese prvog programa u ROM-u, što može doznati iz skripte (početna adresa ROM-a uvećana za veličinu jezgre), uzeti podatke programa da bi došla do drugog – koji se nalazi iza prvog i tako dalje do zadnjeg.

U nastavku su prikazane skripte za jezgru i jedna koju mogu koristiti svi programi.

Više programa u sustavu – skripta za jezgru

```
OUTPUT_FORMAT ( "elf32-i386" )
ENTRY ( kernel_init )

/* konstante */
ROM_START  = 0x01000000;
RAM_START  = 0x03000000;

SECTIONS {
    .kernel_code ROM_START : AT ( ROM_START )
    {
        *( .text* )      /* instrukcije */
        *( .rodata* )    /* konstante */
    }

    kernel_data_in_ROM = .;

    .kernel_data RAM_START : AT ( kernel_data_in_ROM )
    {
        *( .data* )
        *( .bss* )
        . = ALIGN (4096); /* poravnaj kraj na 4 KB */
    }

    programs_in_ROM = kernel_data_in_ROM + SIZEOF (.kernel_data);

    kernel_heap = RAM_START + SIZEOF (.kernel_data);
}
```

Više programa u sustavu – skripta za programe (svaki se zasebno izgrađuje)

```
OUTPUT_FORMAT ( "binary" )
ENTRY ( PROG_INIT )

/* konstante */
PROG_START = 0x00100000;

SECTIONS {
    program_code_in_ROM = 0;
```

```
    /* fizička adresa nije unaprijed poznata jer se programi učitavaju
     * odmah iza jezgre, jedan po jedan, počevši na adresi
     * "programs_in_ROM" */

    .program_code PROG_START : AT (program_code_in_ROM)
    {
        *( .text* ) /* instrukcije */
        *( .rodata* )   /* konstante */
        . = ALIGN (4096);
    }

    program_data_in_ROM = program_code_in_ROM + SIZEOF (.program_code);

    program_data_in_RAM = PROG_START + SIZEOF (.program_code);

    .program_data (program_data_in_RAM) : AT (program_data_in_ROM)
    {
        *( .data* ) /* inicijalizirani podaci */
        *( .bss* )  /* neinicijalizirani podaci */
        . = ALIGN (4096);
    }

    PROG_SIZE = program_data_in_ROM + SIZEOF ( .program_data );
    /* ukupna veličina spremničkog prostora koja je potrebna u ROM-u
     * za ovaj program */

    program_heap = program_data_in_RAM + SIZEOF (program_data);

    program_stacks = program_heap + PROGRAM_HEAP_SIZE;
}
```

S obzirom na to da ima više programa, u ROM će trebati staviti sliku jezgre te sve slike programa iza nje. Slika D.4. prikazuje izvođenje takvog sustava, nakon pokretanja prvog i trećeg programa.

Slika D.4. Moguće zauzeće ROM-a i RAM-a tijekom rada više programa

Isti rezultat dobio bi se i uz korištenje samo jedne skripte koja ima posebne odjeljke za programe 1, 2 i 3.

Primjer D.4. Ugradbeni sustav s različitim spremnicima

Neki sustav ima ROM na adresi `0x10000`, RAM na adresi `0x100000` te brži RAM na adresi `0x10000000`. Slika cijela sustava se početno treba zapisati u ROM. Pri pokretanju kod iz `kernel/boot.c` treba sve podatke (`.data` i `.bss`) prekopirati u RAM osim iz datoteka `kernel/boot.c` i `kernel/core.c`. Sadržaj datoteke (sve odjeljke) `kernel/boot.c` treba staviti na početak ROM-a (i pripremiti za tu adresu), dok sadržaj datoteke (sve odjeljke) `kernel/core.c` treba staviti u ROM, ali pripremiti za brži RAM (kod iz `kernel/boot.c` će to prekopirati kad se sustav pokrene). Osim navedenih datoteka, izvorni kodovi se nalaze i u drugim datotekama i direktorijima (ne samo u `kernel/`). Napisati skriptu za povezivača.

Skripta povezivača

```
OUTPUT_FORMAT ( "elf32-i386" )
ENTRY ( kernel_boot )
ROM = 0x10000
RAM1 = 0x100000
RAM2 = 0x10000000

SECTIONS
{
    .text ROM : AT(ROM)
    {
        *kernel/boot.o (*)
        *(EXCLUDE_FILE (*kernel/core.o) .text)    //iz svih datoteka
        *(EXCLUDE_FILE (*kernel/core.o) .rodata)  //osim core.o
    }
    copy_data_from = ROM + SIZEOF(.text);
    copy_data_to = RAM1;
    .data copy_data_to : AT(copy_data_from)
    {
        *(EXCLUDE_FILE (*kernel/core.o) .data)
        *(EXCLUDE_FILE (*kernel/core.o) .bss)
    }
    data_size = SIZEOF(.data);
    copy_core_from = copy_data_from + data_size;
    copy_core_to = RAM2;
    .core copy_core_to : AT(copy_core_from)
    {
        *kernel/core.o (*)
    }
    core_size = SIZEOF(.core);
}
```

Dodatak E - Nadogradnja sinkronizacijskih mehanizama

Sinkronizacijski mehanizmi prikazani u 11.4. su najčešće dovoljni za ostvarenje višedretvenih programa u "normalnim" sustavima. Međutim, za sustave za rad u stvarnom vremenu i za ugrađene sustave, tj. sustave koji traže veću kontrolu programa ti se mehanizmi dodatno proširuju i dodatnim operacijama i dodatnim mogućnostima.

Od dodatnih operacija u nastavku su opisani mehanizmi *ProbajČekati*, *ČekajKratko* i zauzimanje i otpuštanje više od jednog sredstva (nad istim semaforom ili nad skupom semafora), a od dodatnih mogućnosti *rekurzivno zaključavanje* te metode za rješavanje *problema inverzije prioriteta*. Također su kratko prikazana i tri dodatna mehanizma sinkronizacije: *radno čekanje*, *barijera* i *zaključavanje čitaj-piši*.

E.1. Dodatne operacije sinkronizacije

E.1.1. Operacija *ProbajČekati*

Operacije *Čekaj* mogu pozivajuću dretvu zaustaviti jako dugo – ovisi o stanju sustava, tj. o tome što rade druge dretve. Kada takav nedeterminizam nije prihvatljiv mogu se koristiti operacije *ProbajČekati* ili *ČekajKratko*.

Operacija *ProbajČekati* ispituje stanje sinkronizacijskog mehanizma. Ako je on prolazan, tada je operacija *ProbajČekati* jednaka običnoj operaciji *Čekati* (normalno zauzimanje sredstva). Ako je sinkronizacijski mehanizam bio neprolazan u trenutku poziva *ProbajČekati*, tada dretva neće biti zaustavljena (kao u običnoj operaciji *Čekaj*), samo će po povratnoj vrijednosti operacije (`-1`) i kodu greške (`EAGAIN`) moći doznati da sredstvo ovim pozivom nije zauzeto. Drugim riječima, operacija *ProbajČekati* je inačica obične operacije *Čekaj* koja ne zaustavlja dretvu.

Gotovo svi sinkronizacijski mehanizmi imaju bar jednu takvu operaciju (npr. semafori sa `sem_trywait`, monitori sa `pthread_mutex_trylock`). U nastavku je prikazana operacija *ProbajČekatiSemafor* koja je nastala promjenom izvorne operacije *ČekajSemafor* opisane u 11.4.

```
j_funkcija ProbajČekatiSemafor ( id )
{
    ako ( Sem[id].v > 0 )
    {
        Sem[id].v = Sem[id].v - 1;
        kod_greške = 0;
        povratna_vrijednost = 0;
    }
    inače {
        kod_greške = EAGAIN;
        povratna_vrijednost = -1;
    }
}
```

E.1.2. Operacija *ČekajKratko*

Ponekad zaustavljanje dretve na sinkronizacijskom mehanizmu može biti dozvoljeno, ali samo ako traje kratko, tj. unaprijed poznatog najduljeg intervala vremena. Kada bi zaustavljanje trebalo biti duže dretvu treba propustiti iako nije zauzela sredstvo. Navedenu funkcionalnost pruža operacija *ČekajKratko* (npr. za semafore `sem_timedwait`, za monitore `pthread_mutex_timedlock` i `pthread_cond_timedwait`). Idući pseudokod pokazuje moguće ostvarenje takve operacije nad mehanizmom semafora.

Dodavanje operacije *ČekajKratko* zahtjeva malu promjenu i operacije *Postavi* radi brisanja alarma koji se postavlja prethodnom operacijom, kao i dodavanje operacije za prekidanje čekanja u slučaju da je zadani interval čekanja istekao i dretva je još uvijek zaustavljena.

```
j_funkcija ČekajSemaforKratko ( id, kratko )
{
    kod_greške = 0;
    povratna_vrijednost = 0;

    ako ( Sem[id].v > 0 )
    {
        Sem[id].v = Sem[id].v - 1;
    }
    inače {
        postavi_alarm_za_prekidanje ( prekini_čekanje, kratko,
                                      Aktivna_dretva );
        stavi_u_red ( Sem[id].r, Aktivna_dretva );
        odaberi_aktivnu_dretvu();
    }
}

j_funkcija PostaviSemafor ( id )
{
    ako ( red Sem[id].r je prazan )
    {
        Sem[id].v = Sem[id].v + 1;
    }
    inače {
        prva = uzmi_prvu_iz_reda ( Sem[id].r );
        stavi_u_red ( Pripravne_dretve, prva );
        provjeri_i_obriši_alarm_za_prekidanje ( prva );
        odaberi_aktivnu_dretvu();
    }
}

j_funkcija prekini_čekanje ( dretva ) /* aktivacija alarma */
{
    obriši_alarm_za_prekidanje ( dretva );

    postavi_kod_greške ( dretva, ETIMEDOUT );
    postavi_povratnu_vrijednost ( dretva, -1 );

    makni_dretvu_iz_reda ( dretva, dretva->red );
    stavi_u_red ( Pripravne_dretve, prva );
    odaberi_aktivnu_dretvu();
}
```

E.1.3. Zauzimanje i otpuštanje više od jednog sredstva

Semafor je često suviše jednostavan sinkronizacijski mehanizam. Često je potrebno zauzeti više sredstava, a ne samo jedno (što radi *ČekajSemafor*).

Kada je potrebno zauzeti više sredstava koja su zaštićena istim semaforom, primjerice kad je potrebno koristiti više susjednih elemenata međuspremnika, tada su potrebne operacije koje će atomarno (u jezgrenim funkcijama) smanjiti ili povećati semafor za vrijednost veću od jedan. Neka se takve operacije nazovu *ČekajSemN(id,m)* i *PostaviSemN(id,n)*.

Kada je potrebno zauzeti više sredstava zaštićena različitim semaforima, tada je potrebna operacija koja će atomarno obavljati više operacija nad skupom semaforima. Neka se takve operacije nazovu *ČekajSemafore(sem[m])* i *PostaviSemafore(sem[n])*.

Ostvarenje operacije *ČekajSemN* može biti gotovo identično običnoj *ČekajSemafor*, jedino što se vrijednost semafora ne uspoređuje s jedan već zadanom vrijednošću (drugi parametar funkcije

ČekajSemN(id,broj)). Problem tada postaje kako ostvariti operaciju *PostaviSemN*? Problem je u tome što bi u toj jezgrenoj funkciji trebali imati sve informacije: koje su sve dretve u redu i za koliko one žele smanjiti vrijednost semafora. Kada bismo imali te informacije (zahtjeva dosta promjena u jezgri), operacija *PostaviSemN* bi mogla propuštati redom dretve koje bi sada (nakon povećanja) mogle smanjiti vrijednost semafora. Pitanje je jedino je li stati s propuštanjem na prvoj dretvi za koju se ne može smanjiti semafor ili ići dalje (dok je vrijednost semafora veća od nule) i tražiti dretve koje smanjuju semafor za manju vrijednost.

Drugo ostvarenje operacije *ČekajSemN* može biti suprotno prethodnom – pri zaustavljanju dretve zapisati kod greške (`EAGAIN` – ponovi operaciju), dok u operaciji *PostaviSemN* odmah povećati vrijednost semafora za zadanu vrijednost te propustiti sve zaustavljene dretve. Zaustavljene bi dretve po propuštanju trebale (bar) još jednom pozvati *ČekajSemN* (prethodno bi kao povratnu vrijednost dobile grešku).

Ostvarenje operacije zauzimanja više semafora – *ČekajSemafore(sem[n])* zahtjeva prvo provjeru mogućnosti obavljanja svih operacija nad svim semaforima te potom samo obavljanje operacije. Ako se sve operacije ne mogu obaviti dretvu treba zaustaviti. S obzirom na to da bilo koji od semafora može zaustaviti dretvu, ostvarenje može staviti dretvu u red bilo kojeg neprolaznog semafora i onda slično drugom rješenju za *ČekajSemN* po prvom propuštanju ponovno pozvati istu funkciju ili pak koristiti neko drugo rješenje (značajno drukčije od prikazanog rješenja za semafore). Operacije *PostaviSemafore(sem[m])* povećava vrijednosti za sve semafore iz skupa (*sem[m]*). Ostvarenje te operacije ovisi o ostvarenju prethodne *ČekajSemafore*.

Sučelja na *UNIX* sustavima za rad s takvim semaforima jesu funkcije `semop`, `semget` i `semctl`. Slična sučelja postoje i na *Win32* sustavima preko poziva `WaitForMultipleObjects`.

E.2. Rekurzivno zaključavanje

Uz semafore i monitor pojavljuje se još jedan problem – *rekurzivno zaključavanje*, tj. što napraviti ako dretva koja je već zaključala semafor ili monitor opet pokuša zaključati isti objekt (semafor ili monitor). Ponekad je takvo ponašanje donekle opravdano.

Na primjer, ako se iz početne funkcije koje je ušla u monitor pozivaju druge koje i same imaju zaštitu od paralelnog pozivanja korištenjem istog monitora, kao u primjeru:

```
funkcija_1 ()
{
    Zaključaj_monitor (m);
    nešto_radi;
    funkcija_2();
    još_radi;
    Otključaj_monitor (m);
}
```

```
funkcija_2 ()
{
    Zaključaj_monitor (m);
    nešto_drugo_radi;
    Otključaj_monitor (m);
}
```

Izlaziti iz monitora da bi se u njega opet ušlo može osim nepraktičnosti biti i loše (logički neispravno) rješenje. Problem se može riješiti na nekoliko načina. Jedan od njih je ugradnja podrške za rekurzivno zaključavanje u same sinkronizacijske funkcije.

Podrška rekurzivnom zaključavanju zahtjeva stvaranje sinkronizacijskog objekta u kojemu će to ponašanje biti prihvatljivo. Također, ako se rekurzivno zaključavanje dozvoljava, potrebno je brojati broj rekurzivnih zaključavanja. Ako se dretvi dozvoli da višestruko "uđe" u monitor, ona isto toliko puta mora izaći prije nego li se monitor može smatrati slobodnim. Navedeno se ponašanje može uključiti u jezgrene funkcije za ostvarenje semafora i monitora. Teoretski bi se oni mogli uključiti i u ostale sinkronizacijske mehanizme, ali kod njih način korištenja nije toliko pravilan kao za semafor i monitor te bi pokušaj praćenja svih mogućih stanja mogao rezultirati znatno složenijom izvedbom (u koju se znatno jednostavnije uvuču greške). Dodatni problem rekurzivnog zaključavanja kod monitora jest u tome što osim funkcija za ula-

zak i izlazak iz monitora postoje i druge funkcije, `Čekaj_u_redu` za zaustavljanje dretve i `Propusti_iz_reda` za propuštanje dretve. Operacija zaustavljanja dretve u monitoru mora biti popraćena privremenim izlaskom dretve iz monitora bez obzira na trenutni broj rekurzivnog zaključavanja. Kada se ista dretva propusti, broj rekurzivnih zaključavanja treba ponovno aktivirati. Drugim riječima, rekurzivno zaključavanje napraviti da se koristi u samo funkcijama za ulazi i izlaz iz monitora a ne i u zaustavljanju dretve u monitoru (u redu uvjeta).

Drugi (preporučeni) način zahtjeva promjenu u izvornim kôdovima tako da se operacije u pseudokôdu označene s `nešto_drugo_radi` ostvare u zasebnoj funkciji koja nije zaštićena monitorom te da se poziva iz obje funkcije, tj. iz `funkcija_1` i `funkcija_2` (unutar monitora).

POSIX mehanizam koje omogućuje rekurzivno zaključavanje jest mehanizam monitora (funkcijama `pthread_mutex_lock/pthread_mutex_unlock`) uz prikladnu inicijalizaciju takvog objekta (korištenje konstante `PTHREAD_MUTEX_RECURSIVE`).

E.3. Inverzija prioriteta

U ugrađenim sustavima određenost, pouzdanost i jednostavnost mnogo su značajniji od prosječne učinkovitosti. Dretve u ugrađenim sustavima uglavnom se međusobno razlikuju po važnosti, odnosno prioritetu. Pri dodjeli procesora dretva većeg prioriteta ima prednost ispred one manjeg prioriteta. Iako je takav ili slični slučaj i u ostalim sustavima, odnosno ostalim operacijskim sustavima koji nisu namijenjenih ugrađenoj uporabi, kod njih je prioritet uglavnom nešto što određuje koliko će pojedina dretva dobiti procesorskog vremena, a ne kada će ta dretva postati aktivna. U operacijskim sustavima za rad u stvarnom vremenu i za ugrađene sustave jasno je definirano da kada dretva višeg prioriteta postaje spremna za izvođenje ona istiskuje dretvu nižeg prioriteta koja se trenutno izvodi. Međutim, i u takvim se sustavima događaju slučajevi kada dretva nižeg prioriteta zaustavi izvođenje dretve višeg prioriteta, odnosno dolazi do problem inverzije prioriteta.

Problem *inverzije prioriteta* nastaje kada dretva višeg prioriteta za nastavak rada treba sredstvo koje je zauzela druga dretva nižeg prioriteta. Slika E.1. prikazuje tri slučaja koja se mogu pojaviti u višedretvenom sustavu.

Slika E.1. Problem inverzije prioriteta

Slika E.1. (i) prikazuje uobičajeno ponašanje kada su u sustavu dvije dretve različitog prioriteta. Aktiviranjem prioritetnije dretve, manje prioritetna se istisne, tj. makne s procesora. Na slici, čim dretva A postane spremna istisne dretvu B, koja ima manji prioritet.

Slika E.1. (ii) pokazuje sličnu situaciju, ali dretva A i B tijekom rada koriste zajedničko sredstvo i to međusobno isključivo (na primjer, zaštićeno binarnim semaforom). Dretva manjeg prioriteta B za vrijeme svog rada, dok je A bila neaktivna, zauzela je sredstvo. Kasnije se dretva A aktivirala te odmah istisnula dretvu B. Međutim, u jednom trenutku dretvi A za nastavak rada treba sredstvo koji dretva B još nije otpustila (binarni semafor je neprolazan). Dretva A se zaustavi te dretva B, kao jedina dretva u sustavu nastavlja s radom. Problem koji je nastao naziva se problem inverzije prioriteta jer dretva manjeg prioriteta radi, dok dretva većeg prioriteta čeka na nju. Međutim, čim dretva B oslobodi zauzeto sredstvo, dretva A se otpusti i odmah zauzima sredstvo i nastavlja s radom (primjerice B pozove *PostaviSemafor*).

Slika E.1. (iii) pokazuje situaciju kad u sustavu osim dretvi A i B postoji i treća dretva C prioriteta većeg od dretve B, ali manjeg od dretve A. Kao što se vidi iz grafa, ovakva dretva može dodatno odgoditi izvođenje dretve A. U sustavima s više dretvi vrlo je teško procijeniti koliko se dretva A može odgoditi. Zato je problem inverzije prioriteta vrlo opasan za sustave za rad u stvarnom vremenu!

Problem se u raznim slučajevima rješava na razne načine. Kada je sustav dovoljno određen i poznate su potrebe za sredstvima svih dretvi u sustavu, problem se može izbjeći tako da se dretvi nižeg prioriteta privremeno onemogući zauzimanje sredstava koja će uskoro biti potrebna dretvi višeg prioriteta.

Metode koje se najčešće koriste u slučajevima problema inverzije prioriteta ne rješavaju sam problem nego ublažavaju njegove posljedice. To rade tako da se dretvi koja je zauzela sredstva potrebna prioritetnijoj dretvi (korištenjem sinkronizacijskog mehanizma) omogući što brži rad do oslobađanja dotičnih sredstava. Dva najpoznatija takva protokola su *protokol nasljeđivanja prioriteta* (engl. *priority inheritance protocol*) i *protokol stropnog prioriteta* (engl. *priority ceiling protocol*). Važna pretpostavka za oba protokola je da dretve nakon što sredstvo zauzmu isto će i osloboditi nakon nekog vremena.

E.3.1. Protokol nasljeđivanja prioriteta

Kod *protokola nasljeđivanja prioriteta* koristi se mehanizam "nasljeđivanja prioriteta" i to kada se prioritetna dretva zaustavi na sredstvu koje trenutno zauzima dretva nižeg prioriteta. Toj se drugoj dretvi koja ima stredstvo, privremeno podigne prioritet na razinu prve dretve – dretva nasljeđuje njen prioritet. To se zbiva u samoj funkciji sinkronizacije koju poziva prioritetnija dretva (u pozivu koji zaustavlja dretvu višeg prioriteta). Pri otpuštanju sredstva, dretvi se prioritet vraća na prijašnju vrijednost te konačno dretva višeg prioriteta zauzima sredstvo i nastavlja s radom.

Ostvarenje protokola nasljeđivanja prioriteta zahtjeva mogućnost povećanja prioriteta dretve, ali i pohranu prethodne vrijednosti. S obzirom na to da se dretvi prioritet može i više puta mijenjati, ta struktura mora biti dinamička, slična stogu. Također, povećanje prioriteta može biti i tranzitivno: najprije se poveća prioritet dretve koja drži sredstvo, a potom i prioritet dretve koja zaustavlja prethodnu radi drugog sredstva itd.

Promjena prioriteta može/mora biti popraćena dodatnim operacijama. Ako je red u kojem se dretva nalazi uređen prema prioritetima, tu dretvu treba postaviti na mjesto koje joj sada pripada (možda na prvo mjesto u redu ili bliže njemu). Slične se operacije trebaju raditi i pri povratku prijašnjeg prioriteta dretvi.

Funkcija za jednostruko nasljeđivanje prioriteta, od trenutno aktivne dretve koja se zaustavlja na sredstvu, na dretvu koja je sredstvo prije nje zauzela, a koja se poziva iz funkcije sinkronizacije (primjerice `ČekajSemafor(S)`), može izgledati kao u nastavku (`S` je sredstvo,

`prioritet` je prioritet zaustavljene dretve, `dretve` je opisnik dretve koja je prije zauzela sredstvo, dretva koja nasljeđuje prioritet zaustavljene dretve):

```
funkcija nasljedni_prioritet_povećaj ( S, prioritet )
{
    dretva = vlasnik_sredstva (S);
    ako ( dretva.prioritet < prioritet )
    {
        pohrani_prioritet ( dretva, S );
        promijeni_prioritet ( dretva, prioritet );
    }
}
```

Kao što je vidljivo, funkciji su potrebni neki podaci sustava o dretvama i njihovim prioritetima, sredstvima i trenutnim vlasnicima tih sredstava. Slovo `S` identificira sredstvo koje je uzrok povećanja prioriteta, a može biti semafor ili monitor ili neki drugi sinkronizacijski mehanizam.

Funkciju nasljeđivanja treba pozvati isključivo ako se dotično sredstvo ne može zauzeti, tj. kada dretva ulazi u red. Funkcija `promijeni_prioritet` mijenja prioritet dretvi. S obzirom na to da dretva može biti u raznim stanjima, promjenu prioriteta treba oprezno napraviti. Primjerice, ako je dretva zaustavljena i njen je opisnik u redu koji je organiziran po redu prispijeća dovoljno je promijeniti vrijednost prioriteta dretve u opisniku. Međutim, ako je dretva pripravna ili aktivna, onda bi najbolje bilo dretvu prvo maknuti iz tog reda, promijeniti joj prioritet, staviti u red pripravnih te na kraju pozvati raspoređivač da odredi iduću aktivnu.

Pri povratku sredstva, dretvi treba vratiti njen prijašnji prioritet, tj. treba pozvati funkciju `nasljedni_prioritet_smanji`, koja slijedi u nastavku, iz funkcije za vraćanje sredstva, primjerice `PostaviSemafor(J)` kod semafora.

```
funkcija nasljedni_prioritet_smanji ( dretva, S )
{
    stari_prioritet = obnovi_prioritet ( dretva, S );
    ako je ( stari_prioritet != dretva.prioritet )
        promijeni_prioritet ( dretva, stari_prioritet );
}
```

Kao i prethodnu funkciju i ovu mora slijediti poziv raspoređivača koji će uzeti u obzir nove prioritete.

Za ostvarenje protokola nasljeđivanja prioriteta (kao i protokola stropnog prioriteta) potrebno je sustavno ostvariti podršku za povećavanje prioriteta ali sa zapisivanjem prijašnje vrijednosti tako da se naknadno prioritet može vraćati i na prijašnje vrijednosti kako je to prikazano na slici E.2. Podatkovna struktura za podršku takvih promjena prioriteta mora biti slična stogu, ali s mogućnošću pretraživanja i promjena svih vrijednosti. To može biti polje ograničene duljine ili lista. U nastavku je prikazana jedna takva struktura koja koristi listu te funkcije koje ju koriste.

```
struktura pohr_prio {
    prioritet;
    sredstvo;
}
/* u opisniku dretve postoji lista s tim elementima: .prioriteti */

funkcija pohrani_prioritet ( dretva, S )
{
    novi_element = zauzmi_novi_element ( struktura pohr_prio );
    novi_element.prioritet = dretva.prioritet;
    novi_element.sredstvo = S;
    dodaj_na_početak ( dretva.prioriteti, novi_element );
}

funkcija obnovi_prioritet ( dretva, S )
{
    prvi = dohvati_prvi ( dretva.prioriteti );
```

```
    dok je ( prvi != 0 )
    {
        ako je ( prvi.sredstvo == S )
            prvi.sredstvo = 0;
        prvi = dohvati_idući ( dretva.prioriteti, prvi );
    }

    prioritet = dretva.prioritet;
    prvi = dohvati_prvi ( dretva.prioriteti );
    dok je ( prvi != 0 )
    {
        ako je ( prvi.sredstvo != 0 )
            vrati prioritet;
        prioritet = prvi.prioritet;
        makni_prvi ( dretva.prioriteti );
        oslobodi_element ( prvi );
        prvi = dohvati_prvi ( dretva.prioriteti );
    }
    vrati prioritet;
}
```

U slučaju višestrukog nasljeđivanja prioriteta funkcija za nasljeđivanje postaje nešto složenija. U toj funkciji se osim dretvi koja drži potrebno sredstvo, prioritet može podići i dretvi koja zaustavlja tu dretvu te tako dalje, dretvi koja zaustavlja prethodnu dretvu itd. Kôd funkcije može biti kao u nastavku:

```
funkcija nasljedni_prioritet_povećaj2 ( S, prioritet )
{
    dretva = vlasnik_sredstva(S);
    dok ( dretva.prioritet < prioritet )
    {
        pohrani_prioritet ( dretva, S );
        promijeni_prioritet ( dretva, prioritet );

        R = dretva.red; //sredstvo na koje čeka 'dretva'
        ako ( R != red pripravnih dretvi )
        {
            dretva_vlasnik = vlasnik_sredstva(R);
            ako ( dretva.prioritet > dretva_vlasnik.prioritet )
            {
                dretva = dretva_vlasnik;
                S = R;
            }
        }
    }
}
```

Metoda višestrukog nasljeđivanja za razliku od jednostavnije inačice unosi nešto nedeterminizma u trajanju izvođenja jezgrene funkcije zbog tranzitivnog povećavanja prioriteta.

Protokol nasljeđivanja prioriteta ne rješava problem potpunog zastoja već se za sprječavanje i rješavanje istog moraju upotrijebiti dodatni algoritmi ili postupci. Primjer rada protokola nasljeđivanja prioriteta prikazan je na slici E.2.

Uz protokol nasljeđivanja prioriteta veže se zanimljivost iz svemirske misije na Mars robotske letjelice *Mars Pathfinder* [Jones, 1997] koja je bila upravljana *VxWorks* operacijskim sustavom. Nakon uspješnog slijetanja i prvih nekoliko dana rada, na sustavu upravljanja počeli su se pojavljivati neobjašnjivi i učestali prekidi. Kako je sustav napravljen da prilikom detekcije greške prekida rad i ponovno pokreće cijeli sustav (mehanizmom nadzornog alarma) sve je do određene mjere i dalje radilo. Greška koja je za nekoliko dana pronađena te potom i uklonjena jest u protokolu nasljeđivanja prioriteta, koji je bio greškom isključen. Problem je nastao kada je prioritetnoj dretvi na dulje vrijeme bio uskraćen pristup sabirnici što je sklop za nadzor inter-

Opis događaja po trenucima:

1. E, kao jedina pripravna dretva započinje s radom
2. E zauzima sredstvo S1 (okomite crte)
3. D započinje s radom i istiskuje E jer ima najveći prioritet (u tom trenutku)
4. C započinje s radom i istiskuje D jer ima najveći prioritet
5. C zauzima sredstvo S2 (vodoravne crte)
6. C treba S1 koji je E zauzeo, E nasljeđuje prioritet od C i nastavlja s radom (s prioritetom od C)
7. B započinje s radom i istiskuje E jer ima najveći prioritet
8. A započinje s radom i istiskuje B jer ima najveći prioritet
9. A treba S2 koji je C zauzeo, C nasljeđuje prioritet od A, E nasljeđuje prioritet od C i nastavlja s radom (s prioritetom od A)
10. E oslobađa S1, vraća mu se početni prioritet, C zauzima S1 i nastavlja s radom (s prioritetom od A)
11. C oslobađa S2, vraća mu se početni prioritet, A zauzima S2 i nastavlja s radom
12. A oslobađa S2 i nastavlja s radom
13. A završava, B nastavlja s radom (ima najveći prioritet)
14. B završava, C nastavlja s radom (ima najveći prioritet)
15. C oslobađa S1 i nastavlja s radom
16. C završava, D nastavlja s radom (ima najveći prioritet)
17. D završava, E nastavlja s radom (jedina dretva)
18. E završava

Slika E.2. Primjer korištenja protokola nasljeđivanja prioriteta

pretirao kao kritičnu grešku te je zaustavio i ponovno pokrenuo sustav. Uključivanje protokola nasljeđivanja prioriteta riješilo je problem.

E.3.2. Protokol stropnog prioriteta

Za protokol stropnog prioriteta postoje dvije inačice: *izvorni protokol stropnog prioriteta* i *pojednostavljeni protokol stropnog* prioriteta. Pojednostavljeni protokol ima još nekoliko naziva: *izravni protokol stropnog prioriteta* (engl. *immediate ceiling priority protocol*), *protokol zaštite prioritetom* (engl. *priority protect protocol*) kod POSIX norme te *oponašanje protokola stropnog prioriteta* (engl. *priority ceiling emulation*) kod programskog jezika Java.

Izvorni protokol stropnog prioriteta

Izvorni protokol ima za cilj i izbjegavanje nastajanja potpunog zastoja. Načelna ideja protokola jest da kada dretva želi zauzeti neki semafor tada ona mora imati veći prioritet od najveća stropna prioriteta nekog od već zauzetih semafora. Ako nema takav prioritet, onda joj se ne dopušta da zauzme semafor iako je on možda i u prolaznom stanju. Definicija protokola koja slijedi zasniva se na opisu iz [Rajkumar, 1991].

Osnovne pretpostavke:

1. Razmatra se skup dretvi $\{\mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_N\}$ koje u dijelovima svog izvođenja koriste razna sredstva zaštićena binarnim semaforima $\{\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_M\}$. Za svaku je dretvu unaprijed poznato koje semafore će možda trebati tijekom rada, odnosno za svaki je semafor poznato

koje ga dretve mogu koristiti. Za svaki se semafor izračunava stropni prioritet $p(S_i)$ tako da se odabere najveća vrijednost među prioritetima dretvi koje koriste taj semafor.

2. Koristi se prioritetno raspoređivanje – pripravna dretva najvećeg prioriteta se izvodi.

Rad protokola:

1. Dretva $\mathcal{D}_i$ poziva *ČekajSemafor(S_x)*:
   a) Niti jedan semafor nije zauzet – svi su prolazni.
      - Dretva $\mathcal{D}_i$ zauzima semafor S_x i nastavlja s radom (poziv radi uobičajeno).
   b) Semafor S_x je već zauzet od dretve $\mathcal{D}_j$:
      - Poziv blokira dretvu $\mathcal{D}_i$ te dretva $\mathcal{D}_j$ nasljeđuje prioritet od $\mathcal{D}_i$: $p(\mathcal{D}_j) = \max\{p(\mathcal{D}_j), p(\mathcal{D}_i)\}$.
   c) Semafor S_x nije zauzet, ali neki drugi semafori jesu:
      - Neka S^* označava semafor najvećeg stropnog prioriteta $p(S^*)$ koji je zaključan od strane dretve $\mathcal{D}_k$.
      - Ako je $\mathcal{D}_k = \mathcal{D}_i$ ili $p(\mathcal{D}_i) > p(S^*)$ tada dretva $\mathcal{D}_i$ zauzima semafor S_x i nastavlja s radom.
      - Ako je $p(\mathcal{D}_i) \leq p(S^*)$ poziv će blokirati dretvu $\mathcal{D}_i$ te dretva $\mathcal{D}_k$ nasljeđuje prioritet od $\mathcal{D}_i$: $p(\mathcal{D}_k) = \max\{p(\mathcal{D}_k), p(\mathcal{D}_i)\}$.
2. Dretva $\mathcal{D}_i$ poziva *PostaviSemafor(S_x)*.
   a) Ako je dretvi $\mathcal{D}_i$ bio povećan prioritet radi korištenja semafora S_x, prioritet joj se vraća na prijašnju vrijednost.
   b) Semafor S_x se postavlja u prolazno stanje.
   c) Ako ima dretvi blokiranih na semaforu S_x onda se za najprioritetniju od njih obavljaju akcije označene točkom 1., tj. kao da ta dretva ponovno poziva *ČekajSemafor(S_x)*.

Analizom ovog protokola može se ustanoviti sličnost s protokolom nasljeđivanja prioriteta. I kod ovog se protokola prioritet zaustavljene dretve nasljeđuje u trenutku zaustavljanja. Osnovna je razlika što se u nekim slučajevima dretvi neće dozvoliti zaključavanje semafora iako je on slobodan, a zbog mogućeg zaustavljanja dretvi većeg prioriteta nad drugim semaforima. Na prvi pogled to zaustavljanje izgleda nepotrebno, ali ono zapravo pridonosi rješavanju problema potpunog zastoja, barem osnovnih oblika potpunog zastoja. U slučajevima gdje je problem potpunog zastoja značajan i često se pojavljuje, ovaj protokol može biti dobar odabir (iako bi u takvom sustavu bilo bolje koristiti monitore).

Primjena navedenog protokola u sinkronizacijskim funkcijama zahtjeva posebnu pažnju. Za razliku od protokola nasljeđivanja prioriteta i ako je sredstvo slobodno treba provjeriti smije li zadatak zauzeti sredstvo.

Slika E.3. prikazuje primjer pojavljivanja potpunog zastoja kao i njegovog izbjegavanja korištenjem protokola stropnog prioriteta.

Semafor S^* iz prethodnog opisa algoritma za navedeni primjer prema slici E.3. je zapravo je semafor koji štiti sredstvo X (primjerice X može biti semafor). Stropni prioritet sredstva X jednak je prioritetu dretve A, jer je poznato da će ta dretva i to sredstvo trebati.

U idejnom ostvarenju protokola, funkcija `provjera_ulaza` se poziva prije zauzeća sredstva (semafor, monitor), ako je ono slobodno (ako nije dretva se zaustavlja, kao i inače).

(i) bez korištenja protokola stropnog prioriteta
(ii) uz korištenje protokola stropnog prioriteta

Slika E.3. Primjer potpunog zastoja i kako ga stropni protokol izbjegava

```
funkcija provjera_ulaza ( pri )
{
    za i=1 do broj_sredstava
        ako ( (vlasnik[i] != NITKO) && (VP[i] >= pri) )
            vrati ULAZ_ZABRANJEN;

    vrati ULAZ_DOZVOLJEN;
}
```

Parametar `pri` označava prioritet pozivajuće dretve, `vlasnik[i]` označava dretvu koja je zaključala sredstvo `i` te `VP[i]` stropni prioritet sredstva `i` (prioritet dretve najvećeg prioriteta koja koristi to sredstvo).

Ako dretva većeg prioriteta ne može zauzeti sredstvo jer ga ima dretva nižeg prioriteta, tada se dretvi nižeg prioriteta povećava prioritet i to se radi tranzitivno, ako je potrebno. Funkcije za povećavanje i smanjivanje prioriteta identične su funkcijama prikazanim za protokol nasljeđivanja prioriteta.

Pojednostavljeni protokol stropnog prioriteta

Kod *pojednostavljenog protokola stropnog prioriteta* dretvi se odmah pri zauzimanju sredstva podigne prioritet na unaprijed izračunatu stropnu vrijednost. Na taj se način za vrijeme korištenja nekog sredstva dretvama povećava prioritet da bi one što prije završile s njegovim korištenjem i oslobodile ga za dretve višeg prioriteta. Protokol je jednostavniji za ostvarenje od prethodnog, ali je po pitanju učinkovitosti lošiji. Dretva nižeg prioriteta zauzećem određenog sredstva dobiva veći prioritet i istiskuje prioritetnije dretve čak i onda kada od dretvi višeg prioriteta ne postoji trenutna potreba za sredstvom. Funkcija `podignuti_prioritet` može se pozivati u funkciji sinkronizacije pri zauzimanju sredstva. Najjednostavniji oblik tih funkcija dan je u nastavku.

```
funkcija podigni_prioritet ( S, dretva )
{
    dretva.pohranjeni_prioriteti[S] = dretva.prioritet;
    promijeni_prioritet ( dretva, Prioriteti[S] );
}
```

```
funkcija spusti_prioritet ( S, dretva )
{
    dretva.prioritet = pohranjeni_prioriteti[S];
    promijeni_prioritet ( dretva, dretva.pohranjeni_prioriteti[S] );
}
```

U elementu polja `Prioriteti[S]` pohranjeni su prioriteti koje dretve poprimaju kada zauzmu sredstvo `S`. Navedene funkcije neće ispravno raditi ako dretve zauzimaju više sredstava te se ta sredstva ne oslobađaju obrnutim redoslijedom od redoslijeda zauzeća. Na primjer, ako dretva zauzima redom sredstva S_1, S_2, S_3 i S_4 tada, da bi gornji algoritam ispravno radio, potrebno je sredstva oslobađati redoslijedom: S_4, S_3, S_2 i S_1.

Općenitiji te nešto složeniji algoritam koji radi u općem slučaju zahtjeva dodatnu podatkovnu strukturu istu kao i kod protokola nasljeđivanja prioriteta:

```
funkcija podigni_prioritet ( S, prioritet )
{
    pohrani_prioritet ( dretva, S );
    promijeni_prioritet ( dretva, Prioriteti[S] );
}
```

Smanjivanje prioriteta obavlja se na isti način kao i kod protokola nasljeđivanja prioriteta te se koristi i ista funkcija `nasljedni_prioritet_smanjiti`.

Kao i kod prethodnog protokola obje funkcije sinkronizacije (za zauzimanje i za oslobađanje sredstva) treba slijediti poziv raspoređivača dretvi. Raspoređivač može biti i dio samih funkcija sinkronizacije.

E.3.3. POSIX sučelje za rješavanje problema inverzije prioriteta

POSIX definira mogućnost korištenja protokola nasljeđivanja prioriteta te pojednostavljenog protokola stropnog prioriteta na mehanizmu monitora, tako da se prije inicijalizacije monitora postave odgovarajuće zastavice attributa `attr` kojim se objekt monitora inicijalizira.

```
int pthread_mutexattr_setprotocol ( pthread_mutexattr_t *attr,
                                    int protocol );
```

Protokol se odabire drugim parametrom. Mogućnosti su:

- `PTHREAD_PRIO_NONE` – bez korištenje ijednog protokola
- `PTHREAD_PRIO_INHERIT` – korištenje nasljeđivanja prioriteta
- `PTHREAD_PRIO_PROTECT` – korištenje stropnog prioriteta.

Kada se koristi zadnja opcija, `PTHREAD_PRIO_PROTECT`, tada treba definirati i stropni prioritet pridijeljen monitoru korištenjem funkcije:

```
int pthread_mutex_setprioceiling ( pthread_mutex_t *mutex,
                           int prioceiling,
                           int *old_ceiling );
```

E.4. Ostali sinkronizacijski mehanizmi

Navedeni sinkronizacijski mehanizmi (semafori, monitori, radno zaključavanje) omogućuju sve oblike sinkronizacije. Međutim, ponekad se sinkronizacija može jednostavnije ostvariti korištenjem drugih sinkronizacijskih mehanizama kao što su *radno čekanje*, *barijera* i *zaključavanja čitaj/piši*. Stoga su i ova tri mehanizma ukratko opisana u nastavku (iako nisu ostvarena u Benu).

E.4.1. Radno čekanje

Ponekad (vrlo rijetko) se *radno čekanje* može koristiti i za sinkronizaciju dretvi. Primjerice, u nekom višeprocesorskom sustavu u kojem se gotovo isključivo izvodi jedan višedretveni program, dretve tog programa mogu se međusobno sinkronizirati i radnim čekanjem, izbjegavajući skupe pozive jezgri. Dobitak u brzini će se ostvariti jedino ako te dretve obavljaju poslove koji traju gotovo identično. U protivnom je bolje koristiti sinkronizaciju preko jezgre (semaforima, monitorima ili slično).

Operacije sinkronizacije za radno čekanje ostvaruju se izvan jezgre. Razmotrimo ostvarenje kritičnog osječka radnim čekanjem (engl. *spin-lock*). POSIX definira sučelje za takvu sinkronizaciju imena `pthread_spin_lock` i `pthread_spin_unlock` pa će se ista imena i ovdje koristiti (na primjer, mogla bi se postaviti u `api/pthread.c`).

```
1  typedef volatile unsigned int pthread_spinlock_t;
2
3  int pthread_spin_init ( pthread_spinlock_t *lock, int pshared )
4  {
5          *lock = 0;
6
7          return 0;
8  }
9
10 int pthread_spin_lock ( pthread_spinlock_t *lock )
11 {
12     /* Načelno:   while ( *lock )
13      *               ;
14      *            *lock = 1;
15      *
16      * Problem gornjeg rješenja je u paralelnom čitanju iste
17      * vrijednosti od strane više dretvi - to mora biti atomarno
18      *
19      * Iduće rješenje koristi atomarnu operaciju zamjene sadržaja,
20      * instrukciju "xchg" (http://en.wikipedia.org/wiki/Spinlock)
21      */
22     asm volatile ( "    mov   %0, %%ebx         \n"
23                    "1:  mov   $1, %%eax         \n"
24                    "    xchg  %%eax, (%%ebx)    \n"
25                    "    test  %%eax, %%eax      \n"
26                    "    jnz   1b                \n"
27                    : "=m" (lock) );
28
29     return 0;
30 }
31
32 int pthread_spin_unlock ( pthread_spinlock_t *lock )
33 {
34     *lock = 0;
35
36     return 0;
37 }
```

Zaštita kritičnog odsječka navedenim sučeljem prikazana je primjerom.

```
1  ...
2  pthread_spinlock_t k1;
3  ...
4          /* inicijalizacija */
5          pthread_spin_init ( &k1, 0 );
6          ...
7          pthread_spin_lock ( &k1 );
8          ...
9          /* kritični odsječak */
10         ...
```

```
11          pthread_spin_unlock ( &k1 );
12          ...
```

E.4.2. Barijera

Barijera je sinkronizacijski mehanizam koji se često koristi kada dretve obavljaju iterativni posao kod kojeg je potrebno da se prije pokretanja iduće iteracije dovrše svi poslovi prethodne iteracije. Za sinkronizaciju barijerom potrebno je znati koliko ukupno ima dretvi koje se barijerom sinkroniziraju te inicijalizirati sinkronizacijski objekt. Svaka će dretva po završetku svog posla stati na barijeri – mehanizam će ju zaustaviti. Tek kada dođe zadnja dretva, tada se sve dretve otpuštaju.

Za ostvarenje mehanizma barijere potrebne su dvije osnovne operacije: *DohvatiBarijeru(N)* te *ČekajNaBarijeri(b)*. Skica druge prikazana je u nastavku.

```
j_funkcija Čekaj_na_barijeri ( b )
{
    Bar[b].na_barijeri++;

    ako ( Bar[b].na_barijeri < Bar[b].br_dretvi )
    {
        /* nisu sve dretve još stigle => čekaj */
        stavi_u_red ( Bar[b].r, Aktivna_dretva );
        odaberi_aktivnu_dretvu();
    }
    inače
    {
        /* sve su dretve stigle; ovo je zadnja; propusti sve ostale */
        dok ( red Bar[b].r nije prazan )
            stavi_u_red (Pripravne_dretve, uzmi_prvu_iz_reda(Bar[b].r));

        Bar[b].na_barijeri = 0;
        odaberi_aktivnu_dretvu ();
    }
}
```

POSIX sučelja za barijeru su `pthread_barrier_init` te `pthread_barrier_wait`.

E.4.3. Zaključavanje čitaj-piši

Mehanizam zaključavanja *na* (ili *za*) *pisanje* ili *čitanje* (engl. *read-write locks*) je često potreban u sustavima koji imaju zajedničke podatke koje često samo čitaju, a rijetko mijenjaju. Ideja je da se dretvama koje samo čitaju omogući paralelni rad nad zajedničkim podacima, ali uz osiguranje da dok god se čitanje ne otključa, zajednički podaci se neće promijeniti. Svakim novim zaključavanjem na čitanje mora se ažurirati i brojač čitača. Svakim završetkom čitanja, tj. otključavanjem čitanja, brojač se smanjuje za jedan. Tek kad brojač dođe do nule može se zaključati na pisanje. Svaka promjena zajedničkih podataka treba biti obavljena samo nakon zaključavanja na pisanje. Zaključavanje na pisanje treba biti kritičan odsječak – samo jedna dretva može zaključati na pisanje u nekom trenutku.

Osnovne operacije ovog mehanizma su:

- *ZaključajNaČitanje(čpk)*
- *ZaključajNaPisanje(čpk)*
- *OtključajČitanjePisanje(čpk)*

gdje je *čpk* identifikator sinkronizacijskog čitaj/piši ključa (objekta).

Operacija *ZaključajNaČitanje* će zaustaviti dretvu čitača, ako je sinkronizacijski objekt zaključan

na pisanje. Uobičajeno je da se nova dretva čitača zaustavi i u slučaju kada dretva pisača čeka, iako je možda trenutno sinkronizacijski objekt zaključan na čitanje. U protivnom, moglo bi se dogoditi izgladnjivanje dretvi pisača, neodređeno dugo (i jako dugo) zadržavanje u redu.

Operacija *ZaključajNaPisanje* će zaustaviti dretvu ako se bilo što događa sa sinkronizacijskim objektom (bilo da se čita ili piše).

Operacija *OtključajČitanjePisanje* smanjuje brojač čitača, ako je sinkronizacijski objekt trenutno zaključan na čitanje, odnosno potpuno otključava objekt ako je bio zaključan na pisanje. Naravno, umjesto otključavanja, sinkronizacijski će se objekt odmah zaključati na pisanje ili čitanje, ako u tim redovima postoje dretve koje čekaju. Ovaj mehanizam treba dva reda za zaustavljene dretve: jedan za čitače i jedan za pisače.

POSIX sučelja za zaključavanje čitaj-piši su `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock` te `pthread_rwlock_unlock`.

E.5. Nedorečenosti u korištenju sinkronizacijskih mehanizama

Do sada su razmatrani *očekivani scenariji* uporabe sinkronizacijskih mehanizama. Što je s *neočekivanima*, odnosno koji su to scenariji i kako podesiti ponašanje dretvi i sinkronizacijskih mehanizama u njima?

Jedan od neočekivanih scenarijia uključuje završetak dretve koja je prethodno zauzela ali ne i oslobodila neka sredstva. Primjerice, dretva je ušla u monitor, ali prije nego li je iz njega izašla završila je s radom. Ako se nikakve akcije ne poduzmu pri završetku dretve glede tog monitora, niti jedna druga dretve više neće moći ući u monitor i najvjerojatnije će nastati potpuni zastoj. To je možda i opravdano ponašanje ako je prva dretva završila zbog kritične greške u sustavu nakon koje nema smisla nastaviti s radom (ni u drugim dretvama). Međutim, ako se žele uključiti nekakvi mehanizmi oporavka od pogreške moglo bi se pri micanju dretve iz sustava pogledati koja je sredstva ona koristila i možda neka od njih osloboditi. Za tako nešto je potrebna dodatna struktura podataka koja će pratiti zauzeća te dretve (na primjer, svaki put pri zauzeću novog sredstva dodati opis takvog događaja u listu koja bi onda poslužila za oslobađanje zauzetih sredstava pri završetku dretve). Sličan je pristup pri korištenju datoteka – sve se one zatvaraju pri kraju rada procesa.

Drugi scenariji mogu biti uzrokovani nedostacima sredstava na razini sustava. Primjerice, što ako nema dovoljno slobodnog spremnika za stvaranje nove dretve ili novog sinkronizacijskog objekta? Očekivano, dretva koja to primijeti treba prekinuti svoj radi. Ali što je s ostalim dretvama koje možda očekuju nešto (signal, poruku ili slično) od ove prethodne koja je završila zbog greške? Iako se ovakve pojave događaju vrlo rijetko (ili se ne događaju), u nekim okolinama i njih treba razmotriti i uspostaviti postupke za takve situacije.

Dretvu zaustavljenu na sinkronizacijskim objektom može prekinuti signal, ali i ne mora. Za detalje je potrebno pogledati opis svake sinkronizacijske funkcije. Primjerice, `pthread_mutex_lock` signal prekida, ali se nakon obrade signala dretva ponovno zaustavlja i vraća čekanju na taj monitor. S druge strane, `sem_wait` će signal prekinuti – nakon obrade signala dretva neće više čekati na semafor već nastavit s radom, uz oznaku greške postavljenu na `EINTR` te uz vraćanje vrijednosti `-1` iz funkcije `sem_wait`.

Dodatak F - Raspoređivanje dretvi u operacijskim sustavima

F.1. Raspoređivanje dretvi u stvarnim sustavima

Teoretsko razmatranje raspoređivanja može se pogledati u [SRSV, 2012] (od kuda je preuzet dio teksta za ovaj dodatak). Tamo su prikazane metode:

- raspoređivanje prema mjeri ponavljanja
  - statička pridjela prioriteta, raspoređivanje prema prioritetu
- raspoređivanje prema krajnjim trenucima dovršetaka
  - dinamičko raspoređivanje tijekom rada sustava zadataka
- opće raspoređivanje
  - teoretsko razmatranje optimalnog raspoređivanja skupa zavisnih zadataka na višeprocesorskim sustavima
- raspoređivanje sa stablenom strukturom
  - slično kao i opće raspoređivanje samo malo jednostavnije i manje optimalno
- raspoređivanje prema najmanjoj labavosti
  - slično kao dinamičko raspoređivanje tijekom rada sustava zadataka.

Neki od navedenih postupaka spadaju u klasu statičkih raspoređivača (raspoređivanje prema mjeri ponavljanja, opće raspoređivanje, raspoređivanje sa stablenom strukturom) koji odluke o značajkama raspoređivanja (njihove vrijednosti) donose prije pokretanja sustava, dok drugi postupci spadaju u klasu dinamičkih postupaka (značajke koje se koriste za raspoređivanje mijenjaju se tijekom rada zadataka kao i samim protokom vremena).

U praksi se nastoje preuzeti dobra svojstva pojedinih postupaka raspoređivanja, ali tako da sam postupak bude primjenjiv. S obzirom na to da je vrlo mali broj obilježja zadataka unaprijed poznat, koriste se jednostavniji postupci. Međutim, to ovisi o sustavu koji se razmatra. Negdje su i oni složeniji neophodni, ali ipak vrlo rijetko.

Raspoređivanje značajno ovisi o zahtjevima sustava. Sustavi za rad u stvarnom vremenu samo su jedna skupina sustava u kojoj se koriste računala. U druge skupine možemo ubrojati ugrađene sustave, osobna računala, radne stanice, poslužiteljska računala, prijenosna i mobilna računala (telefoni, dlanovnici, ...). Svojstva zadataka u drugim sustavima nisu jednaka te identični način raspoređivanja dretvi ne mora biti najbolji. Primjerice, multimedijalni program ima svojstva slična zadacima sustava za rad u stvarnom vremenu: svako kašnjenje može izazvati osjetnu degradaciju kvalitete slike ili zvuka. Slično je i s korisničkim sučeljem: znatnija kašnjenja u reakciji na korisničke naredbe korisniku će umanjiti kvalitetu sustava. Ipak, u oba navedena primjera loše raspoređivanje neće izazvati znatne štete (osim smanjenja kvalitete sustava prema ocjeni korisnika). Operacije koje dulje traju, kao što su matematički proračuni, kompresija podataka i prijenos datoteka i ako se još malo odgode neće gotovo ništa umanjiti kvalitetu sustava s obzirom na to da korisnik već očekuje njihovo produljeno trajanje. Zadržimo se za početak ipak samo na zadacima u sustavima za rad u stvarnom vremenu.

Pri ostvarenju sustava, jedan zadatak se preslikava u jednu dretvu te se u nastavku koristi termin dretva umjesto zadatka, odnosno govori se o raspoređivanju dretvi.

F.2. Raspoređivanje dretvi prema prioritetu

Većina operacijskih sustava pripremljena ili prilagođena za sustave za rad u stvarnom vremenu zapravo koristi samo jedan način raspoređivanja – raspoređivanje prema prioritetu. Zato je način dodjele prioriteta dretvama najčešće prema postupku mjere ponavljanja. Odstupanja od tog postupka se koriste kada različite dretve obavljaju poslove različita značaja. Tada arhitekt sustava može nekim dretvama pridijeliti i veći ili manji prioritet od onog koji bi dretva dobila prema postupku mjere ponavljanja.

Samo raspoređivanje – određivanje trenutne dretve za izvođenje obavlja se pri radu sustava tako da se u svakom trenutku među dretvama spremnim za izvođenje (*pripravne dretve*) odabire ona najvećeg prioriteta (koja tada postaje *aktivna dretva*). Dretve koje nisu spremne za izvođenje (*blokirane dretve*) se ne razmatraju pri raspoređivanju.

S obzirom na to da postoji mogućnost da u nekom trenutku najveći prioritet nema samo jedna pripravna dretva već više njih mora se koristiti i dodatni kriterij pomoću kojeg će se odabrati samo jedna dretva. Najčešće korišteni dodatni kriteriji su:

1. prema redu prispijeća (engl. *first in first out – FIFO*) te
2. podjela procesorskog vremena – kružno posluživanje (engl. *round robin – RR*).

S obzirom na to da je prvi kriterij uvijek prioritet, takvi postupci raspoređivanje dobivaju ime prema drugom kriteriju. U kontekstu raspoređivača koji se koriste u stvarnim operacijskim sustavima ime "raspoređivanje prema redu prispijeća" se zapravo odnosi na raspoređivanje kojemu je prvi kriterij prioritet, a tek drugi kriterij je red prispijeća, koji se koristi jedino kada prvi kriterij daje više od jedne dretve. Slično vrijedi i za "raspoređivanje podjelom vremena".

Slika 11.2. prikazuje mogući izgled strukture podataka raspoređivača koji koristi prioritet za raspoređivanje dretvi. Odabir aktivne dretve obavlja se među pripravnim dretvama najvećeg prioriteta, među dretvama A i B. S obzirom na to da je dretva A prva u listi ona će biti odabrana kao aktivna. Ako je drugi kriterij FIFO dretva A će se izvoditi dok ne završi ili dok se ne blokira. Ako je drugi kriterij RR onda će dretvu A u njenom izvođenju prekinuti raspoređivač nakon što je ona "potrošila" svoj dodijeljeni dio procesorskog vremena (kvant), dretva A bit će stavljena na kraj reda dretvi s istim prioritetom, tj. iza dretve B. Iduća aktivna dretva bit će dretva B.

U višeprocesorskim sustavima red pripravnih dretvi (koje se raspoređuju) može biti ostvaren kao na slici 11.2., ali i drukčije. Kod jednog od pristupa za se svaki procesor stvara poseban red pripravnih, prema slici 11.2. Tada postaje potrebno osigurati da se uvijek izvode dretve najveća prioriteta. Ponekad će to zahtjevati "guranje" dretvi u druge redove pripravnih dretvi ili posizanje za dretvama iz redova drugih procesora. Primjerice, kada aktivna dretva A na procesoru I omogući nastavak drugoj dretvi B nastavak svog rada, tada se B može gurnuti procesoru J ako on izvodi dretvu čiji je prioritet manji od dretvi A i B, odnosno i od svih drugih aktivnih dretvi. Slično je i kada neka dretva završi s radom ili se blokira: tada je potrebno odabrati pripravnu dretvu najveća prioriteta među svim pripravnim dretvama – ne samo iz reda procesora koji je sada postao slobodan. U ovom se slučaju radi o "povlačenju" dretve iz reda pripravnih dretvi drugih procesora. Opisani postupci koriste se pri raspoređivanju SRSV dretvi u Linuxu.

Slika F.1. prikazuje primjere raspoređivanja korištenjem prioriteta, kružnog posluživanja te događaja povezanih sa sinkronizacijskim funkcijama.

F.3. Raspoređivanje prema krajnjim trenucima završetaka

Raspoređivanje prema krajnjim trenucima završetaka je vrlo rijetko podržani način raspoređivanja dretvi, čak i među operacijskim sustavima za rad u stvarnom vremenu. U sustavima u kojima jest podržano takvo raspoređivanje, mehanizam njegova korištenja je takav da se po-

Slika F.1. Primjeri raspoređivanja prema prioritetu i podjeli vremena

sebnim sučeljem dretva označi kao periodička, sa zadanom periodom kojom se dretva budi i pokreće, uzimajući u obzir i druge slične dretve i njihove trenutke krajnjih dovršetaka.

Periodičke dretve se mogu prikazati pseudokodom:

```
periodička_dretva
{
   ponavljaj {
      odradi_periodički_posao;
      odgodi_izvođenje ( ostatak_periode );
   }
}
```

Pseudokod takvih dretvi treba proširiti odgovarajućim pozivima da se željeno ostvari. Uobičajeno sučelje koje nude operacijski sustavi koji podržavaju takvo raspoređivanje prikazano je na primjeru:

```
periodička_dretva
{
   označi_periodičnost ( period );
   ponavljaj {
      čekaj_početak_periode ()
      odradi_periodički_posao;
   }
}
```

Operacijski sustav nudi sučelje koje je u primjeru prikazano funkcijama `označi_periodičnost` i `čekaj_početak_periode`.

F.4. Podrška za raspoređivanje dretvi prema POSIX sučelju

POSIX [POSIX] je zajednički naziv za skupinu IEEE standarda kojima se definira sučelja koja operacijski sustavi trebaju pružati programima, a radi njihove prenosivosti. Prenosivost programa je glavni razlog nastajanja POSIX-a i sličnih standarda. U početku su ciljani operacijski sustavi bile razne inačice UNIX-a, ali se kasnije sučelje počelo širiti tako da obuhvaća i sustave za rad u stvarnom vremenu. Službena oznaka standarda je IEEE 1003, a naziv međunarodnog standarda je ISO/IEC 9945. Standardi su nastali iz projekta koji je počeo oko 1985. Naziv POSIX predložio je Richard Stallman, a naknadno je izvedeno značenje (engl. *backronym*) *portable operating system interface* (prenosivo sučelje operacijskih sustava), pri čemu X predstavlja UNIX.

POSIX standard ostvaruju mnogi operacijski sustavi, pogotovo oni predviđeni za sustave za rad u stvarnom vremenu. U kontekstu SRSV-a POSIX definira nekoliko načina raspoređivanja (klase dretvi):

1. `SCHED_FIFO` – po redu prispijeća,
2. `SCHED_RR` – kružno posluživanje,
3. `SCHED_SPORADIC` – sporadično raspoređivanje te
4. `SCHED_OTHER` – raspoređivanje nekritičnih dretvi.

Raspoređivanja `SCHED_FIFO` i `SCHED_RR` ostvaruju se na način prikazan u F.2.

Raspoređivanje `SCHED_SPORADIC` je noviji postupak raspoređivanja (vrlo rijetko još podržan u operacijskim sustavima, primjerice [QNX, 6.3]), pripremljen za periodičke dretve, kod kojih dretva tijekom svake svoje periode dobiva kratko vrijeme izvođenja s višim prioritetom. Ako u tom kratkom vremenu ne obavi svoj periodički posao, prioritet joj se smanjuje tako da suviše ne utječe na ostale dretve sustava. Parametri tog raspoređivanja su: period obnavljanja prioriteta (engl. *replenishment period*), povlašteno vrijeme rada (engl. *initial budget*), povlašteni prioritet (engl. *high priority*) te smanjeni prioritet (engl. *lower priority*). Primjer rada ovog raspoređivača prikazuje slika F.2.

Sporadično raspoređivanje je namijenjeno periodičkim poslovima većeg prioriteta koji uglavnom kratko obave svoje operacije (u svakoj pojavi/periodi). Međutim, ponekada se može dogoditi da posao zahtjeva malo više vremena. Da se u tim slučajevima ne bi narušili ostali poslovi, `SCHED_SPORADIC` će za te dulje obrade smanjiti prioritet dretvi, nakon početnog povlaštenog vremena s većim prioritetom.

Način `SCHED_OTHER` koristi se za dretve koje nisu kritične, koje se ne koriste za upravljanje. Njihovo raspoređivanje opisano je u F.5.

Postavljanje algoritma raspoređivanja i prioriteta dretvi može se obaviti funkcijom:

```
int pthread_setschedparam ( pthread_t thread,
                            int policy,
                            const struct sched_param *param );
```

Samo mijenjanje prioriteta može se obaviti i s pozivom:

```
int pthread_setschedprio ( pthread_t thread,
                           int prio );
```

Ekvivalentne funkcije na razini procesa su `sched_setscheduler` i `sched_setparam` (s

Slika F.2. Primjer sporadičnog raspoređivanja

istim ili ekvivalentnim parametrima).

Ponekad je jednostavnije prije stvaranja nove dretve definirati njene parametre za raspoređivanje. Stvaranje dretve obavlja se funkcijom:

```
int pthread_create ( pthread_t *thread,
                     const pthread_attr_t *attr,
                     void *(*start_routine)(void*),
                     void *arg );
```

Drugi parametar funkcije (`attr`) definira atribute za novu dretvu te ga je potrebno postaviti prema željenim svojstvima nove dretve, prije stvaranja nove dretve. Pozivom:

```
int pthread_attr_setinheritsched ( pthread_attr_t *attr,
                                   int inheritsched );
```

definira se nasljeđuje li nova dretva parametre raspoređivanja od dretve koja ju stvara (kada je drugi parametar `PTHREAD_INHERIT_SCHED`) ili ih treba zasebno postaviti (kada je drugi parametar `PTHREAD_EXPLICIT_SCHED`).

Postavljanje načina raspoređivanja te prioriteta u `attr` obavlja se s pozivima:

```
int pthread_attr_setschedpolicy ( pthread_attr_t *attr,
                                  int policy);
int pthread_attr_setschedparam ( pthread_attr_t *attr,
                                 const struct sched_param *param );
```

Prioritet dretvi se može promijeniti i pod utjecajem sinkronizacijskih mehanizama. To je objašnjeno uz opis sinkronizacijskih mehanizama u E.3.

Ako se u sustavu dretve dinamički stvaraju prema potrebama, onda je potrebno razmatrati i njihove završetke. Naime, svaka dretva zauzima dio sredstava sustava. Pri završetku dretve sva zauzeta sredstva se ne oslobađaju automatizmom, već neka ostaju zauzeta. Sredstva se u potpunosti oslobađaju kad je status dovršene dretve preuzet od neke druge dretve (npr. pozivom `pthread_join`) ili ako je dretva pri stvaranju označena kao odvojena (engl. *detachable*). Označavanje dretve kao odvojive se postavlja u atributu `attr` pozivom `pthread_attr_setdetachstate` ili to radi sama dretva pozivom `pthread_detach`.

Stvaranje dretvi koje se raspoređuju po metodama `SCHED_FIFO` i `SCHED_RR` zahtjeva povlaštenog korisnika (administratora) budući da njegove dretve mogu u potpunosti istisnuti sve druge dretve pa čak i one operacijskog sustava.

F.5. Raspoređivanje nekritičnih dretvi

Pri raspoređivanju nekritičnih dretvi mogu se koristiti razna načela:

- pravednost – pravedno raspodijeliti procesorsko vrijeme svim dretvama u sustavu (u skladu njihovim prioritetima)
- veća procesorska iskoristivost – uz višu iskoristivost više će se posla obaviti
- veći broj završenih dretvi – dovršiti što više dretvi – favorizirati kratke dretve
- minimizacija čekanja u redovima – smanjuje se prosječno vrijeme koje dretve provedu u redu prije nego li su poslužene (npr. smanjiti kvant vremena)
- kraće vrijeme odziva – dati prednost interaktivnim dretvama – one rijetko koriste procesor pa im zato odmah posvetiti pažnju čim budu pripravne (obično upravljaju s korisničkim sučeljem ili UI jedinicama)
- optimizacija korištenja priručnih spremnika – u sustavima s više procesorskih jedinki zadržavati dretvu na istom procesoru jer u sukcesivnim izvođenjima (nakon zamjene s drugim dretvama) te dretve mogu pronaći svoje podatke još uvijek u priručnom spremniku i time smanjiti vrijeme njihovog dohvata pri radu dretve.

U operacijskim sustavima opće namjene dretve koji nisu vremenski kritične uglavnom se raspoređuju principom pravedne podjele procesorskog vremena. Dretve i tu imaju atribut prioriteta, ali se prioritet koristi za određivanje koliko će procesorskog vremena te dretve dobiti (izračun se obavlja uzimajući sve pripravne dretve i njihove prioritete). Uobičajeni princip raspoređivanja koji se koristi za takve dretve naziva se višerazinsko raspoređivanje s povratnom vezom (engl. *multilevel feedback queue – MFQ*). Ciljevi koje to raspoređivanje nastoji ostvariti su:

- dati prednost dretvama s kratkim poslovima
- dati prednost dretvama koje koriste ulazno-izlazne naprave
- na osnovi rada dretve ustanoviti u koju skupinu dretva pripada te ju prema tome dalje raspoređivati.

Duge dretve, tj. dretve koje su procesorski intenzivne, koje trebaju puno procesorskog vremena (koje bi cijelo vrijeme koristile procesor) se žele potisnuti. Naime, takve dretve ionako duže traju te se očekuje da će kasnije biti gotove te će njihova kratka odgoda zbog izvođenja drugih dretvi manje utjecati na sustav nego odgoda kratkih dretvi, koje, primjerice upravljaju korisničkim sučeljem i čija reakcija mora biti promptna (inače se kod korisnika stvara dojam sporosti sustava).

Sam postupak raspoređivanja koji nastoji poštovati navedena načela može se opisati skupom pravila ponašanja raspoređivača nad skupom dretvi koje su složene prema trenutnim prioritetima u redove prema redu prispijeća (slično slici 11.2.):

- Uvijek se raspoređuje prva dretva iz reda najvećeg prioriteta (najviši neprazni red).
- Procesor se dretvi dodijeljuje za određeni interval vremena (kvant vremena).
- Ako dretva završi u tom intervalu, ona napušta sustav (ne razmatra se više u postupku raspoređivanja).
- Ako se dretva pri svom izvođenju u tom njoj dodijeljenom intervalu blokira, miče se iz reda pripravnih, ali se kasnije, pri odblokiranju stavlja u isti red pripravnih dretvi iz kojeg i otišla

ili ovisno o vremenu provedenom u blokiranom stanju, čak se postavlja i u više redove (podiže joj se prioritet).

- Ako se dretva izvodi cijeli interval i nije ni gotov niti se blokirao za vrijeme tog izvođenja, onda ga raspoređivač prekida i miče u red niže (smanjuje mu prioritet za jedan).
- Postupak se ponavlja dok god ima pripravnih dretvi i dok one ne dođu do najnižeg reda (reda najmanjeg prioriteta). U tom redu se dretve poslužuju podjelom vremena (ne idu u red niže na kraju intervala, već samo na kraj istog reda).
- Kada u sustav dođe nova dretva, ona se stavlja u najviši red (najprioritetniji red), na kraj tog reda.

Opisani postupak vrlo brzo procjenjuje dretvu: je li ona spada u kategoriju kratkih ili dugih. Ako je kratka, ostaje joj prioritet, a ako je duga prioritet joj pada tijekom izvođenja.

Višerazinsko raspoređivanje s povratnom vezom se ne ostvaruje u operacijskim sustavima upravo prema prikazanim pravilima, ali se sličnim pristupima ostvaruju navedena načela.

Razmotrimo primjere raspoređivanja u operacijskim sustavima zasnovanim na jezgri Linuxa inačice 3.0 te sustavima zasnovanim na porodici operacijskih sustava Microsoft Windows.

Primjer F.1. Linux

Operacijski sustavi zasnovani na Linuxu podržavaju POSIX sučelje, ali ne i sve načine raspoređivanja, tj. (još) ne podržavaju `SCHED_SPORADIC`. Prioriteti kritičnih dretvi koje se raspoređuju po `SCHED_FIFO` i `SCHED_RR` kreću se od 1 do 99 (veći broj označava veći prioritet).

Za raspoređivanje nekritičnih dretvi koristi se prioritet 0 te sljedeći načini raspoređivanja:

- `SCHED_OTHER` – uobičajeno raspoređivanje zasnovano na principu pravedne podjele vremena
- `SCHED_BATCH` – za duge dretve (prisutan tek od inačice jezgre 2.6.16)
- `SCHED_IDLE` – za dretve najmanjeg prioriteta (prisutan tek od inačice jezgre 2.6.23).

Nekritične dretve će dobiti procesorsko vrijeme tek kada nema niti jedne kritične dretve u redu pripravnih, tj. dretve koja se raspoređuje prema `SCHED_FIFO` ili `SCHED_RR`.

Dretve koje spadaju u klase `SCHED_OTHER`, `SCHED_BATCH` i `SCHED_IDLE` imaju osnovni prioritet postavljen na nulu (manji od najmanjeg za kritične dretve), ali za međusobnu usporedbu (raspoređivanje) koriste drugu vrijednost, takozvanu *razinu dobrote* (engl. *nice level*) koja se kreće od -20 (najveća) do $+19$ (najmanja). Dobrote ispod nule može postavljati samo administrator (korisnik *root*). Dobrota dretve utječe na to koliko će procesorskog vremena dretva dobiti u odnosu na ostale dretve različite dobrote. Primjerice, dretva s razinom dobrote q trebala bi dobiti 10 do 15 % više procesorskog vremena od dretve razine $q + 1$.

Načini raspoređivnja `SCHED_OTHER` i `SCHED_BATCH` su slični jedino što će raspoređivač uvijek pretpostaviti da je dretva u klasi `SCHED_BATCH` duga dretva i zbog toga biti neznatno u lošijem položaju od ostalih dretvi (u klasi `SCHED_OTHER`).

Dretve u klasi `SCHED_IDLE` imaju najmanju dobrotu i neće se izvoditi dokle god ima drugih dretvi koje nisu u istoj klasi.

Od inačice jezgre 2.6.23 Linux za raspoređivanje nekritičnih dretvi (`SCHED_OTHER`, `SCHED_BATCH` i `SCHED_IDLE`) koristi se novi raspoređivač naziva *potpuno pravedan raspoređivač* (engl. *completely fair scheduler – CFS*). Korištenjem izračunatog virtualnog vremena

koje pripada pojedinoj dretvi i stvarno dodijeljenog vremena, tj. razlike tih vremena izgrađuju se crveno-crna stabla s pripravnim dretvama te se za aktivnu odabire ona dretva kojoj sustav najviše duguje (s najvećom razlikom).

Prikažimo rad potpuno pravednog raspoređivača na pojednostavljenom primjeru.

Neka se u sustavu u početnom trenutku nalazi pet dretvi ista prioriteta $\{D_1, \ldots, D_5\}$. Raspoređivač će odabrati jednu, recimo D_1, i njoj dati kvant vremena T_q. Nakon što taj kvant istekne, raspoređivač će ažurirati virtualna vremena koja pripadaju pojedinim dretvama: u tom intervalu svaka od dretvi dobiva jednaki dio virtualnog vremena, tj. $T_q/5$. S obzirom na to da se samo D_1 izvodila jedino će se njoj povećati dodijeljeno vrijeme za T_q što će uzrokovati pomak te dretvu u stablu – ona više neće biti prva (već zadnja u ovom trenutku). Zato će raspoređivač u idućem trenutku odabrati neku drugu dretvi $\{D_2, \ldots, D_5\}$ kao aktivnu.

Primjer F.2. Microsoft Windows operacijski sustavi

Raspoređivanje u porodicama operacijskih sustava Microsoft Windows (NT, 2000, XP, 2003, Vista, 7, 8, 10) obavlja se korištenjem prioriteta dretvi. Prioritet dretve računa se na osnovu prioritetne klase procesa i prioritetne razine dretvi, prema tablicama F.1. i F.2.

Tablica F.1. Prioritetne klase procesa

IDLE_PRIORITY_CLASS	ID
BELOW_NORMAL_PRIORITY_CLASS	BN
NORMAL_PRIORITY_CLASS	N
ABOVE_NORMAL_PRIORITY_CLASS	AN
HIGH_PRIORITY_CLASS	H
REALTIME_PRIORITY_CLASS	RT

Tablica F.2. Prioritetne razine dretvi

THREAD_PRIORITY_IDLE	ID
THREAD_PRIORITY_LOWEST	L
THREAD_PRIORITY_BELOW_NORMAL	BN
THREAD_PRIORITY_NORMAL	N
THREAD_PRIORITY_ABOVE_NORMAL	AN
THREAD_PRIORITY_HIGHEST	H
THREAD_PRIORITY_TIME_CRITICAL	TC

Tablice F.3. i F.3. prikazuju dodjeljivanje prioriteta na osnovu prioritetnih klasa i razina.

Tablica F.3. Izračun prioriteta za normalne dretve (ID, BN, N, AN, H)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ID	ID	L	BN	N	AN	H									TC
BN	ID			L	BN	N	AN	H							TC
bN	ID				L	BN	N	AN	H						TC
fN	ID						L	BN	N	AN	H				TC
AN	ID							L	BN	N	AN	H			TC
H	ID										L	BN	N	AN	H,TC

Prioritet 0 (najniži prioritet) rezerviran je za posebne dretve operacijskog sustava koje brišu oslobođene stranice (u postupku upravljanja spremnikom straničenjem).

Tablica F.4. Izračun prioriteta za kritične dretve (RT)

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ID	−7	−6	−5	−4	−3	L	BN	N	AN	H	3	4	5	6	TC

Raspoređivanje kritičnih dretvi (klasa REALTIME_PRIORITY_CLASS) se ponešto razlikuje od ostalih klasa, odnosno raspoređivanje je identično prethodno opisanom načinu SCHED_RR, uz raspon prioriteta od 16 do 31.

Raspoređivanje nekritičnih dretvi obavlja se korištenjem njihovih prioriteta – dretve najvećeg prioriteta se odabiru za izvođenje, slično SCHED_RR, ali uz iznimke:

- Prve iznimke su dretve procesa u klasi NORMAL_PRIORITY_CLASS (N) koji je trenutno u fokusu korisnika (engl. *foreground process*) koji dobiva malo povećanje prioriteta (prefiks f u tablici F.3.) u odnosu na dretve koje nisu u fokusu (prefiks b).
- Druge iznimke su pripravne dretve koje dugo nisu dobile ništa procesorskog vremena. Njima raspoređivač povremeno dodijeljuje dio procesorskog vremena da izbjegne njihovo izgladnjivanje i također da ublaži problem inverzije prioriteta.

Dodatak G - Dodavanje novih raspoređivača

U odjeljku 11.1.3. opisan je osnovni način raspoređivanja dretvi prema prioritetu, ali ukratko i drugi (po redu prispijeća, podjelom vremena, prema trenucima krajnjih završetaka). Ugrađivanje drugih raspoređivača u Benu izravno uz prioritetni moglo bi značajno povećati složenost koda. S obzirom na to da je jedna od osnovnih ideja sustava Benu da bude što je moguće jednostavniji u osnovnim elementima (u što spada i osnovno prioritetno raspoređivanje), smišljeno je drukčije rješenje koje će minimalno promijeniti osnovno raspoređivanje (i dodano u pokaznu granu `Chapter_07_Threads/XX_07a_Sched2` koja se koristi u ovom poglavlju).

Ideja proširivosti novim raspoređivačima jest da se u osnovi i dalje koristi početni prioritetni raspoređivač, a da dodatni raspoređivači samo utječu na dretve koje raspoređuju, uglavnom preko njihova prioriteta i stanja. Akcije koje će takvi dodatni raspoređivači poduzimati zbivat će se posebnim trenucima:

- u trenutku aktiviranja dretve, a prije samog povratka u dretvu
- u trenutku micanja dretve s procesora (zbog zaustavljanja te dretve ili zbog pojave dretve veća prioriteta)
- u trenucima aktiviranja alarma koje postavljaju dodatni raspoređivači (ako ih postavljaju).

Dodavanje novih raspoređivača je stoga ostvareno samo dodavanjem dvije linije koda u postojeći raspoređivač (u `kthreads_schedule`): prije povratka u odabranu aktivnu dretvu poziva se operacija *aktiviraj* te prije micanja aktivne dretve poziva se operacije *isključi*.

Isječak kôda G.1. Chapter_07_Threads/XX_07a_Sched2/kernel/sched.c:kthreads_schedule()

```
133         if (!curr || !kthread_is_active(curr) ||
134                 kthread_get_prio(curr) < kthread_get_prio(next))
135         {
136                 if (curr && !kthread_is_passive(curr)) /* deactivate curr */
137                 {
138                         ksched2_deactivate_thread(curr);
139
140                         /* move last active to ready queue, if still ready */
141                         if (kthread_is_active(curr))
142                                 kthread_move_to_ready(curr, LAST);
143
144                         /* deactivation might change ready thread list */
145                         next = get_first_ready();
146                         ASSERT(next);
147                 }
148
149                 /* activate next */
150                 next = kthread_remove_from_ready(next);
151                 ASSERT(next);
152
153                 kthread_set_active(next);
154
155                 ksched2_activate_thread(next);
156         }
```

Obje operacije ovise o dretvi koja se aktivira ili isključuje, odnosno o njenom raspoređivaču. Ako njen raspoređivač ne koristi te operacije one se neće ni pozvati.

Upravljanje dodatnim raspoređivačima ostvareno je u `kernel/sched.c`, ali tako da je u `kernel/sched.h` definirano sučelje koje svaki dodatni raspoređivač mora koristiti. Sami dodatni raspoređivači se ostvaruju u zasebnim datotekama, kao što je to napravljeno i na primjerima kružnog posluživanja (`kernel/sched_rr.*`) te raspoređivanja prema trenucima

krajnjeg završetka (`kernel/sched_edf.*`).

Isječak kôda G.2. Chapter_07_Threads/XX_07a_Sched2/kernel/sched.h

```
118 /*! Secondary scheduler interface */
119 struct _ksched_t_
120 {
121         int  sched_id;
122              /* scheduler ID, e.g. SCHED_FIFO */
123
124         int (*init)(ksched_t *ksched);
125              /* initialize scheduler */
126
127         int (*schedule)(ksched_t *ksched);
128              /* schedule - pick next active thread */
129
130         int (*thread_add)(ksched_t *ksched, kthread_t *kthread,
131                           int sched_priority, sched_supp_t *sched_param);
132         /* actions when thread is created or when it switch to this scheduler */
133
134         int (*thread_remove)(ksched_t *ksched, kthread_t *kthread);
135         /* actions when thread is removed from this scheduler */
136
137         int (*thread_activate)(ksched_t *ksched, kthread_t *kthread);
138              /* actions when thread is to become active */
139
140         int (*thread_deactivate)(ksched_t *ksched, kthread_t *kthread);
141              /* actions when thread stopped to be active */
142
143         int (*set_thread_sched_parameters)(
144                ksched_t *ksched, kthread_t *kthread, sched_supp_t *param);
145              /* set scheduler specific parameters to thread */
146
147         int (*get_thread_sched_parameters)(
148                ksched_t *ksched, kthread_t *kthread, sched_supp_t *param);
149              /* get scheduler specific parameters from thread */
150
151         ksched_params_t  params;
152                          /* scheduler specific data */
153 };
```

Promjena načina raspoređivanja dretve ostvareno je sučeljem `pthread_setschedparam`. Drugi je način da se prije stvaranja dretve postave odgovarajuće vrijednosti u element `pthread_attr_t` koji se predaje funkciji za stvaranje nove dretve i tako stvori dretva sa zadanim načinom i parametrima raspoređivanja. Kada se to ne napravi, koristi se pretpostavljeno raspoređivanje prema prioritetu i dretva dobije pretpostavljeni prioritet (definiran s `THR_DEFAULT_PRIO` u `config.ini`).

G.1. Raspoređivanje podjelom vremena

Raspoređivanje podjelom vremena radi tako da dretve ista prioriteta ravnomjerno dijele procesorsko vrijeme. Ideja je izbijeći izgladnjivanje dretvi u sustavu u kojem je bar jedna dretva *radna* – koja bi koristila procesor duži interval vremena (a ostale bi ju tada trebale čekati).

Svaka dretva povremeno treba dobiti jedan *kvant* vremena (definiran elementom `time_slice` strukture `ksched_rr_t`). Po isteku kvanta dretvu treba zaustaviti, vratiti na kraj reda pripravnih dretvi – reda koji odgovara njenom prioritetu te iz tog reda uzeti prvu dretvu (aktivirati ju).

Za ostvarenje navedenog postupka potreban je jedan alarm koji se postavlja u trenutku aktiviranja dretve – u operaciji *aktiviraj* (`thread_activate` elementu `ksched_t` sučelja raspoređivača podjelom vremena). Po aktiviranju alarma (isteku kvanta), dretva se vraća u red

pripravnih (na kraj svog reda) iz kojeg se potom uzima iduća po redu (prva iz reda).

Međutim, dretvu u njenom izvođenju mogu prekinuti i drugi događaji – prekidi naprava. Takvi prekidi mogu propustiti neku drugu dretvu u red pripravnih, dretvu koja može imati veći prioritet od prekinute (koja se raspoređuje podjelom vremena). U takvim slučajevima treba zapamtiti koji dio kvanta dretva nije iskoristila. Takva se dretva pri micanju s procesora, pri vraćanju u red pripravnih dretvi stavlja na prvo mjesto liste u koju dretva svojim prioritetom spada. Kad prioritetnija dretva završi sa svojim radom, ponovno će se aktivirati dretva čiji je kvant bio prekinut, ali sada će dobiti samo dio kvanta koji joj je preostao.

Navedeno ponašanje je ugrađeno u funkcije `rr_thread_activate/deactivate` i `rr_timer` te ostvarene u `kernel/sched_rr.c`.

Primjer korištenja raspoređivača podjelom vremena prikazan je programom u `programs/round_robin`.

G.2. Raspoređivanje prema trenucima krajnjih završetaka

Kada neka dretva ne bi stigla obaviti svoj posao do tog trenutka sustav ne bi stigao obaviti taj dio posla na vrijeme, tj. sustav ne bi radio ispravno. Raspoređivanje koje u obzir uzima trenutak do kojeg dretve trebaju završiti svoj dio posla je znatno složenije za ostvariti. Vrlo se rijetko susreću operacijski sustavi koji imaju ovaj način raspoređivanja. Uglavnom se problemi raspoređivanja rješavaju povećanjem računalne snage, tj. pristupom povećanja zalihosti procesorske snage. U rijetkim slučajevima to ipak nije dovoljno. S druge strane, za ugrađene sustave je jako nepovoljno ako treba koristiti jače računalo jer će ono trebati više energije (jače napajanje ili baterije).

Raspoređivanje prema trenucima krajnjih završetaka ostvareno u Benu je pokazno, u svrhu prikaza mogućnosti dodavanja novih raspoređivača. U stvarnim primjenama bilo bi bolje raspoređivanje izravno ugraditi i povezati s osnovnim prioritetnim raspoređivačem ili njega potpuno zamijeniti ovim. U nastavku slijedi opis ostvarenog raspoređivača.

Osnovna pretpostavka na osnovu koje je izgrađeno rješenje jest da su dretve koje se raspoređuju ovim raspoređivačom cikličke i da se mogu opisati sljedećim pseudokodom:

```
dretva ()
{
    neki posao koji nema TKZ; /* TKZ = trenutak krajnjeg završetka */
    ...

    postavi_raspoređivanje_prema_TKZ ( period, tkz ); /* edf_set */

    dok je ( neki uvjet )
    {
        čekaj_početak_periode (); /* edf_wait */
        obavi periodički posao;
    }

    makni_raspoređivanje_prema_TKZ (); /* edf_exit */

    neki posao koji nema TKZ;
    ...
}
```

Sve tri operacije: postavljanje raspoređivača, zaustavljanje dretve do iduće periode te micanje iz zadanog raspoređivača obavljaju se jezgrenim sučeljem za promjenu parametara raspoređivanja `pthread_setschedparam` (da se ne proširuje jezgra kad je navedeno sučelje dovoljno). S druge strane, prema programima su pripremljena jednostavnija sučelja (`edf_set`, `edf_wait` te `edf_exit` u `api/pthread.c`).

Raspoređivač ima tri vlastita reda za dretve[1]:

1. `wait` – red dretvi čiji period još nije započeo (one čekaju svoje iduće aktiviranje)
2. `ready` – red pripravnih dretvi čiji je period započeo
3. `active` – red (kazaljka) na aktivnu dretvu, tj. dretvu čiji je trenutak krajnjeg završetka najskoriji, bliži od svih ostalih dretvi iz reda pripravnih.

Gornje operacije upravljaju dretvama – premještaju ih iz redova lokalnog raspoređivača, ali i iz osnovnog: stavljaju ili miču iz reda pripravnih.

Operacija postavljanja raspoređivanja prema trenutku krajnjeg završetka dodaje zadane vrijednosti u opisnik dretve (perioda ponavljanja te trenutak krajnjeg završetka koji se izražava relativnom vrijednošću od početka periode). Potom se dretva postavlja u red pripravnih ovog raspoređivača (`ready`) i miče iz osnovnog raspoređivača (ona je u tom trenutku bila aktivna dretva). Na kraju se pozove interna funkcija raspoređivača (`edf_schedule`) koji će među svim (svojim) dretvama (`active` i sve iz reda `ready`) uzeti onu s najbližim trenutkom krajnjeg dovršetka, proglasiti ju aktivnom (premjestiti u `active`), premjestiti u red pripravnih dretvi osnovna prioritetna raspoređivača i pozvati njega (`kthreads_schedule`).

U operaciji čekanja na početak periode, usporedit će se trenutna vrijednost sata i vrijeme početka periode. Ako je perioda već počela obavit će se dio operacije koji je prethodno opisan (prvo u red `ready` pa `edf_schedule`). Ako perioda još nije počela dretva se miče u red `wait` te se poziva `edf_schedule`.

U operaciji micanja iz ovog raspoređivača, dretva se miče iz njega, postavlja joj se osnovno raspoređivanje (prema prioritetu) te poziva `edf_schedule`.

U prve dvije operacije moraju se postaviti i alarmi koji će buditi dretve na početku njihovih perioda te u trenucima krajnjih dovršetaka (radi provjere ispravnog rada sustava). Navedeni alarmi se aktiviraju funkcijama `edf_period_alarm` i `edf_deadline_alarm`. Alarmi provjeravaju stanje dretve i miču je u odgovarajuće redove, ovisno o stanju (treba ju staviti u `ready` ili ju primjerice prekinuti ako nije stigla odraditi sve do potrebnog trenutka).

Primjer korištenja raspoređivača prema trenucima krajnjih završetaka dan je u `programs/EDF` . Radi ispravne simulacije trošenja procesorskog vremena, a da bi se vidjeli učinci raspoređivanja, primjer je potrebno podesiti prema računalnoj snazi sustava na kojem se pokreće. Ciljano trajanje petlje – radnog čekanja koja se ponavlja `LOOPS` puta je trećina sekunde. Naime, tada će se dogoditi prekoračenje trenutka krajnjeg završetka za bar jednu dretvu (ali ne sve).

[1]Redovi dretvi koje koristi raspoređivač prema trenucima krajnjih završetaka nisu vidljivi osnovnom raspoređivaču (prema prioritetu) - svaki raspoređivač ima svoje redove. Jedino se pri promjeni aktivne dretve u pomoćnom raspoređivaču, traži promjena te dretve u redovima osnovnog raspoređivača (njegovim sučeljem).

Dodatak H - Primjeri programskih zadataka za vježbu

Idući primjeri programskih zadataka služe za upoznavanje s razvojnim alatima i izvornim kodovima sustava, njihovo prilagođavanje i proširivanje.

`Chapter_01_Startup`

1. Pripremiti razvojnu okolinu (postaviti sve potrebne programe na sustav). Ispitati okolinu pokretanjem zadanih primjera.
2. Stvoriti granu u repozitoriju gdje će se pohraniti vježbe.

`Chapter_02_Source_tree`

3. Pokrenuti pripremljeni sustav uz praćenje rada (*debuggiranje*) i dovesti ga do točke _____ (bit će definirano pri predaji).
4. Pokrenuti pripremljeni sustav i otkriti greške u kodu praćenjem rada sustava (sustav će biti zadan pri predaji).
5. Pokrenuti pripremljeni sustav i otkriti greške u kodu korištenjem umetnutih ispisa s `LOG` i `ASSERT` (sustav će biti zadan pri predaji).
6. Odvojiti ispis na zaslon za programe i jezgrene funkcije promjenom upravljačkog programa tako da se gornja polovica zaslona koristi za ispise programa, a donja za jezgrene funkcije. Neka jezgra i programi koriste različito `console_t` sučelje.
7. Proširiti operaciju ispisa tako da se mogu ispisivati brojevi tipa: `float` i `double`.
8. Smjestiti globalne varijable na adresu 0x130000. Ispisati adrese jedne lokalne varijable i jedne globalne varijable. Definirati varijablu `posebna_varijable` koja će biti smještena na adresu 0x200000 te u programu ispisati i njenu vrijednost i adresu.
9. Pripremiti sustav za adresu 0x200000. Jedino funkciju `premjesti()` pripremiti za adresu 0x100000 koja će ostatak sustava (pripremljenog za 0x200000, ali učitanog na 0x100000) prekopirati na 0x20000 te "pokrenuti".

`Chapter_03_Interrupts`

10. Proširiti prekidni podsustav tako da se prekidi obrađuju u skladu s prioritetima koji su zasebno zadani za svaki prekid, tj. prekidni broj. (Obrada prekida se obavlja s dozvoljenim prekidanjem!)
11. Proširiti prekidni podsustav tako da se za svaki prekid može registrirati više prekidnih potprograma proširenjem polja `ihandler` u dvodimenzionalno polje s tri moguća elementa za svaki prekidni broj. (Prilagoditi sve funkcije prekidnog podsustava.)
12. Koristiti APIC za upravljanje prekidima (umjesto sklopa Intel 8259).
13. Ostvariti algoritam dinamičkog upravljanja spremnikom Doug Lee metodom (ili vrlo sličnom).
14. Ostvariti dinamičko upravljanje spremnikom kod kojeg se raspoloživi dio spremnika dijeli u segmente nad kojima zasebno djeluju algoritmi dinamičkog upravljanja spremnikom (*memory pool sustav*).

`Chapter_04_Timer`

15. Napisati program koji simulira upravljanje s 4 aktivnosti koje treba periodički obrađivati: prvu svake sekunde, drugu svake dvije sekunde, treću svake tri sekunde te četvrtu svakih

pet sekundi. Simulirati obradu radnim čekanjem s trajanjem do 500 ms (trajenje simulirati petljom s potrebnim brojem iteracija). Prije početka obrade ispisati poruku o obradi koja će sadržavati i trenutno vrijeme (sat). Isto napraviti i po dovršetku obrade. Promijeniti podsustav upravljanja alarmima tako da se za vrijeme obrade alarma dozvoli prekidanje.

16. Programski ostvariti nadzorni alarm (watchdog timer) u jezgri. Alarm treba imati mogućnost postavljanja te brisanja alarma (limite za prerano i prekasno javljanje, akcija koju poduzeti u slučaju prekoračenja (ALARM, WARNING)).
17. Ostvariti sat `CLOCK_MONOTONIC` (pored postojećeg `CLOCK_REALTIME`).
18. Ostvariti mjerenje vremena (i opcionalno i alarme) korištenjem drugih sklopova (osim korištenog i8253).

`Chapter_05_Devices`

19. Proširiti upravljanje napravama (`sys__dev_send/recv`) tako da pozivi blokiraju ako se operacija ne može trenutno obaviti i zastavica `IPC_WAIT` je postavljena. Kao jedan od mogućih primjera kako to napraviti, pogledati ostvarenje odgode u `kernel/time.c` (s time da se može pojednostaviti i ne registrirati funkcije za prekide ili ipak dodati povratne pozive (`callback`) u naprave).
20. Dodati novi upravljački program za neku napravu koja se trenutno ne koristi ili proširiti postojeći upravljački program (npr. grafički način rada grafičke kartice).
21. Ispitati upravljački program za serijsku vezu. Popraviti ga tako da se može koristiti preusmjeravanje kada se kao emulator koriste programi VMware Player i *VirtualBox* (i slični).
22. Ostvariti cjevovod preko jezgre koristeći isti mehanizam kao i za dodavanje naprava (isto sučelje i principe dodavanja – dodati cjevovod kao napravu).

`Chapter_06_Shell`

23. Razraditi mogućnosti prekidanja i pauziranja programa. Dodati mogućnost prekidanja trenutnog programa s [Ctrl]+[C], pauziranja izvođenja s [Ctrl]+[P] (i nastavljanja istom kombinacijom tipki).
24. Napisati programe `dir [direktorij]` te `cat datoteka` koji ispisuju sadržaje zadanih direktorija i datoteka. S obzirom na to da nema datotečnog sustava, navedene naredbe trebaju simulirati korištenje direktorija i datoteka izvornih kodova. Ne ručno kopirati izvorne kodove (u nekakve nizove znakova). Ispis mora odražavati izvorne kodove koji su se koristili za stvaranje sustava ("ono što se prevelo treba i ispisati"). Koristiti "skripte" koje će pri samom prevođenju izgraditi potrebnu strukturu podataka iz izvornih kodova (ako je potrebno).

`Chapter_07_Threads`

25. Ostvariti upravljanje prekidom tako da se obrada prekida obavlja korištenjem dretvi (pri pojavi prekida zahtjev za obradu staviti u red (listu), a posebne dretve uzimaju zahtjeve iz tog reda i obrađuju prekide prema prioritetu).
26. Proširiti način korištenja naprava tako da se omogući blokiranje dretvi ako se operacija slanja ili primanja podataka ne može odmah dovršiti. Npr. pri zahtjevu za čitanjem s tipkovnice, blokirati dretvu dok se tipka ne stisne; pri komunikaciji sa serijom blokirati dretvu ako je ulazni međuspremnik prazan a pozvana je operacija čitanja, ... Osmisliti globalni koncept i pokazati na primjeru tipkovnice i serije.
27. Proširiti funkcionalnost reda poruka i upravljanje signalima tako da se za svaki tip signala (svaki broj tj. prioritet) može registrirati zasebna funkcija koja ga obrađuje.
28. Dodati sustav detekcije potpunog zastoja (nad zadanim skupom dretvi). Sustav treba do-

biti skup opisnika dretvi koje prati i u slučaju detekcije potpunog zastoja zaustavlja dotične dretve (miče ih iz sustava) i ispisuje prikladnu poruku na konzolu. Potpuni zastoj nastaje ako su sve dretve blokirane na sinkronizacijskim mehanizmima (nisu ni aktivne ni pripravne ni blokirane na UI napravama).

29. Dodati sinkronizacijske funkcije nad mehanizmom monitora tako da omogućuju vremenski ograničeno blokiranje (`monitor_timedlock`).

`Chapter_08_Processes`

30. Proširiti sučelje za definiranje i korištenje privatnog prostora dretve tako da bude slično POSIX sučelju: `pthread_key_create`, `pthread_setspecific` i `pthread_getspecific` prikazanom primjerom u tekstu.
31. Proširiti jezgrene funkcije za ostvarenje monitora mogućnošću rekurzivnog zaključavanja. Predvidjeti da se oslobađanje monitora (privremeno) događa i u `monitor_wait` funkciji te shodno tome brojač rekurzija povezati i s dretvom a ne samo s monitorom. Npr. proširiti opisnik dretve listom dodatnih podataka za sinkronizacijske funkcije koja se provjerava u tim funkcijama (jedan takav element neka broji rekurzivna zaključavanja, ako ih ima).
32. Proširiti slanje i primanje poruka tako da se dodatno za svaki proces stvara vlastiti red u koji je moguće slati poruke i iz njega čitati (oznaka `MSG_PROCESS`). Ponašanje treba biti vrlo slično redu pridjeljenom dretvi. Razlika je što se pri slanju poruka u red treba navesti proces (ili `NULL` ako se šalje u proces pozivajuće dretve). Čitanje dozvoliti samo dretvama pripadnog procesa.
33. Sve pozive jezgri (sklopovske i programske) stavljati u red i obrađivati po redu prispijeća s nekoliko jezgrenih dretvi prethodno pripremljenih za to. Koristiti zadnji inkrement, tj. `Chapter_08_Processes/06_Processes`.
34. Napisati program koji će simulitati rad proizvođača i potrošača. Dretva proizvođača i potrošača za razmjenu podataka trebaju koristiti mehanizam poruka – jednom porukom prenese se cijela informacija. Kraj slanja poruka proizvođač dojavljuje signalom dretvi potrošaču, koja tek po tom signalu čita poruku iz reda poruka. Sadržaj poruka proizvođač treba generirati s tri periodička alarma različitih perioda – svaki u ulazni spremnik stavlja po jedan znak pri aktiviranju. Proizvođač šalje znakove tek kad ih ima 5 u međuspremniku (sinkronizaciju na strani proizvođača napraviti proizvoljno, ali bez radnog čekanja).

Dodatak I - Zadaci za vježbu

U ovom dodatku nalaze se zadaci (s prošlih ispita) koji provjeravaju potrebno znanje iz područja. Neki zadaci imaju i ponuđena rješenja.

---

1. Zadan je makro:

```
#define LOG(LEVEL,FORMAT,...) \
fprintf ( log, #LEVEL FORMAT "\n", ##__VA_ARGS__ )
```

Ako se on pozove s `LOG(A, "%d", a);` u što će se pretvoriti makro u početnoj fazi prevođenja (engl. *preprocessing*)?

Rješenje: `fprintf ( log, "A%d\n", a );`

---

2. Napisati makro `KVJ(A,B,C,X1R,X1I,X2R,X2I)` za izračunavanje rješenja kvadratne jednadžbe oblika: `A*x^2 + B*x + C = 0`. Pretpostaviti da su parametri realni brojevi te da `A` nije nula. Ulazni parametri: `A`, `B` i `C` mogu biti i složeni izrazi pa i povratne vrijednosti funkcija (npr. `KVJ(5+f1(),3,get(a),x1r,x1i,b,c)`).

Rješenje:

```
#define KVJ(A,B,C,X1R,X1I,X2R,X2I)          \
do {                                        \
    double a = (A), b = (B), c = (C);       \
    double korijen = b*b - 4*a*c;           \
    if ( korijen >= 0 ) {                   \
        korijen = sqrt(korijen);            \
        X1R = (-b - korijen) / 2 / a;       \
        X2R = (-b + korijen) / 2 / a;       \
        X1I = X2I = 0;                      \
    }                                       \
    else {                                  \
        korijen = sqrt(-korijen);           \
        X1R = X2R = -b / 2 / a;             \
        X1I = -korijen / 2 / a;             \
        X2I = -X1I;                         \
    }                                       \
}                                           \
while(0)
```

---

3. Napraviti makro `POSTAVI(tip, podaci, duljina, poruka)` koji će popuniti varijablu strukture:

```
struct poruka {
    short tip;
    short duljina;
    char podaci[1];
}
```

Pretpostaviti da postoji sučelje `malloc` za zauzimanje spremnika za novi objekt.

Rješenje:

```
#define POSTAVI(tip, podaci, duljina, poruka)                    \
do {                                                             \
    poruka = malloc ( sizeof(struct poruka) - 1 + duljina );     \
```

```
    poruka->tip = tip;                                          \
    poruka->duljina = duljina;                                  \
    memcpy ( poruka->podaci, podaci, duljina );                 \
}                                                               \
while(0)
```

---

4. Funkcije u nekoj datoteci koriste se i za jezgru i za programe (npr. `memcpy`). Međutim, obzirom na korištenje sklopovlja za pretvorbu logičkih adresa u fizičke za programe, dok jezgra koristi fizičke adrese (apsolutne), isti se kod dva puta prevodi i kasnije zajedno povezuje ('linka'). Stoga se pri prevođenju trebaju koristiti različita imena funkcija (npr. `memcpy` za programe te `kmemcpy` za jezgru). Napisati dio koda koji definira makro `FUNKCIJA(ime_funkcije)` koji treba koristiti pri deklaraciji funkcija. Preko makroa `JEZGRA` može se doznati je li funkciji treba dodati prefiks `k` (kada je `JEZGRA=1`) ili ne. Npr. korištenje makroa bi izgledalo:

```
void * FUNKCIJA(memcpy) ( void *dest, const void *src, size_t num )
```

što se u početnoj fazi prevođenja treba prevesti u:

```
void * memcpy ( void *dest, const void *src, size_t num )
```

ili

```
void * kmemcpy ( void *dest, const void *src, size_t num )
```

Rješenje:

```
#if JEZGRA == 1
    #define FUNKCIJA(ime_funkcije)  k ## ime_funkcije
#else
    #define FUNKCIJA(ime_funkcije)  ime_funkcije
#endif
```

---

5. Zašto sljedeći makroi nisu dobri? Pokazati primjer kada se ne dobiva željeno ponašanje.

```
#define NEG(X)          -X
#define INC_MOD(X,MOD)  X=(X+1)%MOD
```

---

6. Napisati makro `MAX(A,B,C)` koji vraća najveću vrijednost od `A`, `B` i `C`. Makro se koristi za izraze koji su brojevi (tipovi podataka koji se mogu uspoređivati s >, < i ==). Izrazi koji se šalju kao parametri su jednostavni (ne mijenjaju varijable, npr. neće se pozvati s `MAX(a++,b--,c);`).

---

7. Neka funkcija, zadana prototipom:

```
1: int neka_funkcija ( int prvi, char drugi, double treci );
```

poziva se iz programa kodom:

```
2: x = neka_funkcija ( a, b, c ); //a tipa int, b tipa char, c tipa double
```

a) Prikazati instrukcije koje će se generirati pri prevođenju programa, tj. linije 2 (za x86 ili ARM ili slične).

b) Prikazati stanje stoga u trenutku nakon izvođenja poziva potprograma, a prije izvođenja prve instrukcije potprograma.

---

8. Funkcija `int print(char *format, ...)` prima varijabilan broj parametara. Pretpostaviti da se u ostvarenju te funkcije poziva funkcija:

```
int print_dev ( device_t *dev, __________ )
```

prema primjeru ispod, kojoj treba proslijediti iste parametre. Kako to napraviti? Nadopuniti sljedeći kod i komentirati.

```
int printf ( char *format, ... )
{
   device_t *dev = dohvati_stdout (); //funkcija postoji

   return print_dev ( dev, ___________________________________________ );
}
```

---

9. Čemu služe ključne riječi: `extern`, `static`, `inline` i `volatile`? Opisati njihovo korištenje na primjerima.

---

10. Funkcije `op1`–`op6` se ostvaruju datotekama `op.h` i `op.c`. Funkcije `op1` do `op3` se koriste samo unutar datoteke `op.c`, dok funkcije `op4`–`op6` unutar i izvan te datoteke. Funkcije `op2`, `op4` i `op6` koriste globalnu varijablu `var1` definiranu u datoteci `op.c`. Varijabla `var2`, također definirana u `op.c`, se izravno koristi i iz datoteke `op2.c`. Funkcija `op4` poziva se iz obrade prekida. Funkcije `op3` i `op5` su kratke te ih treba ugraditi na mjesto poziva (`inline`). Napisati datoteke `op.h` i `op.c`. Na početku datoteke `op.c` uključuje se zaglavlje `op.h`. Pretpostaviti da su sve funkcije tipa `int opX (int p);` te da sve što nije potrebno izvan datoteke se i ne može koristiti izvan datoteke. Pri deklaraciji varijabli (neka su sve tipa `int`) koristiti dodatno potrebne ključne riječi radi postizanja navedenih zahtjeva. Tijelo svih funkcija zamijeniti s `nešto`.

---

11. Na zadanom kodu označiti koji dio (varijable, kod) će se staviti u koji odjeljak pri prevođenu (koristiti samo `.text`, `.data` i `.bss` odjeljke).

```
#include "zaglavlje.h"                  Rješenje:

struct nesto[10];                       .bss
int a = 3;                              .data
static int b = 5;                       .data

int main ()
{
    int x, y;                           .bss (ili drugdje, na stogu)
    x = a * 5; y = x * a;               .text
    b += funkcija1 ( x, y, nesto );     .text
    a += funkcija2 ( nesto );           .text
    return a + b;                       .text
}
```

---

12. Neki sustav se sastoji od datoteka: `a.h`, `a.c`, `b.h`, `b.c` te `main.c`. Odgovarajuće `.c` datoteke koriste odgovarajuća zaglavlja tj. `.h` datoteke, dok `main.c` koristi oba zaglavlja. Pri prevođenju datoteke `a.c` treba koristiti zastavicu `-DZ1`, za `b.c` zastavicu `-DZ2` te za `main.c` zastavice `-DZ3 -DZ4`. Pri povezivanju (engl. *linking*) treba postaviti zastavicu `-lnesto`. Napisati `Makefile` kojim će se moći izgraditi zadani sustav u program naziva `test1`.

Rješenje:

```
test1: a.o b.o main.o
    gcc -o test1 a.o b.o main.o -lnesto

a.o: a.c a.h
    gcc -c a.c -DZ1

b.o: b.c b.h
    gcc -c a.c -DZ2

main.o: main.c a.h b.h
    gcc -c main.c -DZ3 -DZ4
```

---

13. Neki sustav treba pripremiti za učitavanje u ROM na adresi `0x10000`. Podaci (odjeljci `.data` i `.bss` će se pri pokretanju kopirati na adresu `0x100000` te ih (podatke) treba pripremiti za tu adresu (ali učitati u ROM). Napisati skriptu za povezivača (engl. *linker*) koja će omogućiti navedeno. U skriptu dodati potrebne varijable.

Rješenje:

```
ROM = 0x10000;
RAM = 0x100000;
SECTIONS {
    .kod ROM:
    {
        * ( .text .rodata )
    }
    podaci_pocetak = .;
    .podaci RAM : AT ( ROM + SIZEOF(.kod) )
    {
        * ( .data .bss )
    }
    podaci_kraj = podaci_pocetak + SIZEOF(.podaci);
}
```

---

14. Jedan projekt za ugrađene sustave sastoji se od tri datoteke: `startup.c`, `mkernel.c` i `programs.c` (te njihova zaglavlja, ekvivalentne `.h` datoteke). Pri prevođenju sve će se izlazne (`.o`) datoteke sastojati samo od `.text`, `.rodata`, `.data` i `.bss` odjeljaka. Izlazna datoteka `kernel.bin` koja nastaje povezivanjem sve tri izlazne objektne datoteke (`startup.o`, `kernel.o` i `programs.o`) posebnim se alatom upisuje u stalni spremnik (ROM) na adresi 0x20000. Pretpostaviti da će program u `startup.c` prekopirati podatke jezgre (`.rodata`, `.data` i `.bss` iz `mkernel.o`), instrukcije i podatke programa (sve iz `programs.o`) u radni spremnik (RAM) na adresu 0x100000. Nadalje, pretpostaviti da se koristi neki oblik upravljanja spremnikom (dinamičko ili straničenje) te da se programi nalaze u logičkom prostoru koji započinje s adresom 0x30000.

a) Napisati skriptu `skripta.ld` za povezivača (linkera) tako da navedeno bude moguće (uključujući dodavanje potrebnih varijabli).

b) Napisati `Makefile` za prevođenje projekta uz pretpostavku da su zastavice za prevođenje: `-c -O -m32 -nostdlib` te zastavice za povezivanje: `-O -T skripta.ld -s`.

---

15. Neki ugrađeni sustav sadrži sljedeće spremnike:

1. ROM na adresi 0x100000
2. RAM na adresi 0x200000

3. priručni spremnik na adresi 0xF000000.

Programska potpora sastoji se od jezgre OS-a (u direktoriju `jezgra`) te programa P_1, P_2, ... P_6 koji se nalaze u direktorijima `programi/p1`, `programi/p2`, ...`programi/p6`. Slika sustava će se posebnim alatima spremiti u ROM. Pri pokretanju sustava posebno sklopovlje sve će prvo kopirati u RAM. Tada se treba pokrenuti funkcija `init` (iz datoteke `jezgra/init.c`) koja će sve elemente jezgre premjestiti u priručni spremnik (i od tu se nadalje koristiti). Programi P_1, P_2 i P_3 pokretat će se iz RAM-a (za njega ih treba pripremiti), dok će ostali iz priručnog spremnika, kamo će se kopirati pri pokretanju programa. Programi P_4, P_5 i P_6 pokreću se pojedinačno – nikad nisu dva istovremeno aktivna (ne bi ni stali u priručni spremnik, uz jezgru). Dakle, programe P_4, P_5 i P_6 treba pripremiti za istu početnu adresu u priručnom spremniku (odmah iza jezgre). Sustav nema sklopovlje za dinamičko pretvaranje adresa – sve adrese moraju biti pripremljene za lokacije s kojih će se koristiti. Napisati skriptu za povezivača u koju uključiti sve potrebne varijable.

Rješenje:

```
ROM = 0x100000;
RAM = 0x200000;
CACHE = 0xF000000;

SECTIONS {
    init_ROM = ROM;
    init_RAM = RAM;
    init_run = init_RAM;
    .init init_run : AT ( init_ROM ) {
        jezgra/init.o ( * )
    }
    init_size = SIZEOF(.init);

    kernel_ROM = init_ROM + init_size;
    kernel_RAM = init_RAM + init_size;
    kernel_run = CACHE;
    .kernel kernel_run : AT(kernel_ROM) {
        jezgra* ( * )
    }
    kernel_size = SIZEOF(.kernel);

    p123_ROM = kernel_ROM + SIZEOF(.kernel);
    p123_RAM = kernel_RAM + SIZEOF(.kernel);
    p123_run = p123_RAM;
    .p123 p123_run : AT ( p123_ROM ) {
        programi/p1* ( * )
        programi/p2* ( * )
        programi/p3* ( * )
    }
    p123_size = SIZEOF(.p123);

    p4_ROM = p123_ROM + SIZEOF(.p123);
    p4_RAM = p123_RAM + SIZEOF(.p123);
    p4_run = kernel_run + SIZEOF(.kernel);
    .p4 p4_run : AT ( p4_ROM ) {
        programi/p4* ( * )
    }
    p4_size = SIZEOF(.p4);

    p5_ROM = p4_ROM + SIZEOF(.p4);
    p5_RAM = p4_RAM + SIZEOF(.p4);
    p5_run = kernel_run + SIZEOF(.kernel);
    .p5 p5_run : AT ( p5_ROM ) {
        programi/p5* ( * )
    }
    p5_size = SIZEOF(.p5);
```

```
    p6_ROM = p5_ROM + SIZEOF(.p5);
    p6_RAM = p5_RAM + SIZEOF(.p5);
    p6_run = kernel_run + SIZEOF(.kernel);
    .p6 p6_run : AT ( p6_ROM ) {
        programi/p6* ( * )
    }
    p6_size = SIZEOF(.p6);
}
```

---

16. Neki ugrađeni sustav ima tri naprave. Prve dvije N1 i N2 treba poslužiti iz obrade prekida funkcijama `n1()` i `n2()` (te funkcije postoje), dok se s N3 upravlja programski – u petlji glavnog programa, pozivom `n3()`. Naprava N1 neće ponovno generirati zahtjev za prekid dok prethodni zahtjev te naprave nije obrađen do kraja. Isto vrijedi i za napravu N2. N3 ne generira zahtjeve za prekid. Naprava N1 jest najvažnija i njene zahtjeve treba najmanje odgađati (tj. ne odgađati). Funkcije `n1()`, `n2()` i `n3()` mogu trajati proizvoljno dugo (prema potrebi u pojedinome trenutku). Sustav posjeduje prekidni podsustav sa sučeljem:

```
registriraj_prekid ( id, funkcija );
zabrani_prekidanje ();
dozvoli_prekidanje ();
```

Pokazati ostvarenje funkcija `x_n1()` i `x_n2()` te `glavni_program()` u koje će se postaviti dodatne potrebne operacije prije poziva `n1()`, `n2()` i `n3()` (prema potrebama).

Rješenje:

```
glavni_program ()
{
    registriraj_prekid ( N1, x_n1() );
    registriraj_prekid ( N2, x_n2() );
    dozvoli_prekidanje ();
    ponavljaj {
        n3();
    }
}
x_n1()
{
    n1(); //obrada prekida sa zabranjenim prekidanjem;
}
x_n2()
{
    dozvoli_prekidanje ();
    n2(); //obrada prekida s dozvoljenim prekidanjem;
    zabrani_prekidanje ();
}
```

---

17. Neki procesor ima samo jednu prekidnu liniju (žicu) na koju su spojene sve naprave. Skicirati potrebnu strukturu podataka te jezgrine funkcije za ostvarenje prekidnog podsustava.

---

18. U pseudokodu prikazati ostvarenje prekidnog podsustava. Ostvariti sve neophodne funkcije za korištenje iz jezgre. Rješenje mora biti potpunije, npr. nije dovoljno napisati 'spremni kontekst dretve' već 'spremni kontekst dretve u opisnik dretve'. Pretpostaviti da se pri prihvatu prekida automatski na stog pohranjuje programsko brojilo i identifikator prekida (broj) te da se u programsko brojilo upisuje vrijednost 10. Pretpostaviti da za svaki broj (do `BR_PREKIDA`) postoji samo jedan mogući izvor prekida.

Rješenje:

```
//struktura podataka:
polje_kazaljki_na_funkciju obrada[BR_PREKIDA];

//sučelja za jezgru
inicijalizacija_prekidnog_podsustava()
{
    za i = 0 do BR_PREKIDA-1
        obrada[i] = NULL;
    omogući prekidanje;
}

registriraj_prekid ( id, funkcija )
{
    ako je ( id > 0 && id <= BR_PREKIDA )
        obrada[id] = funkcija;
}

//obrada prekida
10: //na adresi 10
    pohrani sve korisničke registre procesora na stog;
    pozovi "obrada_prekida"
    obnovi sve korisničke registre procesora sa stoga;
    vrati se iz prekida; //učitaj PC, makni id sa stoga, dozvoli prekidanje

obrada_prekida()
{
    x = dohvati_opisnik_aktivne_dretve ();
    kopiraj_kontekst_u_opisnik_dretve;
    id = dohvati_id_prekida_sa_stoga;
    ako je ( obrada[id] != NULL )
        obrada[id] (id);
    x = dohvati_opisnik_aktivne_dretve ();
    kopiraj_kontekst_iz_opisnika_dretve_x_na_stog;
}
```

---

19. Zadan je algoritam dinamičkog upravljanja spremnikom kod kojeg su slobodni blokovi u listi složenoj prema načelu "zadnji unutra – prvi van" (engl. *last-in-first-out – LIFO*), tj. kod kojeg se pri oslobađanju bloka i nakon njegova eventualna spajanja sa susjednim on u listu slobodnih blokova dodaje na početak liste. Navesti dobra i loša svojstva tog algoritma.

---

20. Čemu služi nadzorni alarm (engl. *watchdog timer*)?

---

21. Ostvariti podsustav za upravljanje vremenom sa sučeljem:

```
int dohvati_sat ( int *sek, int *usek );
int postavi_sat ( int sek, int usek );
int postavi_alarm ( int sek, int usek, void (*funkcija)() );
```

Za ostvarenje na raspolaganju stoji brojilo dohvatljivo na adresi `0x8000` koje odbrojava frekvencijom od 1 Mhz. Najveća vrijednost koja stane u brojilo je 10^9. Kada brojilo dođe do nule izazove prekid i stane. U obradi tog prekida pozove se funkcija `prekid_brojila()` (koju treba napraviti, pored gornjih). Vrijeme u sekundama i mikrosekundama je relativno u odnosu na neki početni trenutak (nebitno koji). Kada se mijenja vrijeme s `postavi_sat`, postojeće alarme treba poništiti.

Rješenje:

```
#define MAXCNT 1000000000
#define TICKSPERSEC     1000000 // 1 MHz
```

```
int zadnje_ucitano = MAXCNT;
int *brojilo = (int *) 0x8000;
int sat_sec = 0, sat_usec = 0;
int alarm_sec = 0, alarm_usec = 0;
void (*alarm) () = NULL;

int postavi_sat ( int sek, int usek )
{
    //provjere preskočene ( sek >= 0 && usek >= 0 && usek < 1000000 )
    sat_sec = sek;
    sat_usec = usek;
    alarm = NULL; //poništava se alarm
    zadnje_ucitano = MAXCNT;
    *brojilo = zadnje_ucitano;
    return 0;
}

int dohvati_sat ( int *sek, int *usek )
{
    //provjere preskočene ( sek != NULL && usek != NULL )
    *sek = sat_sec + ( zadnje_ucitano - *brojilo ) / TICKSPERSEC;
    *usek = sat_usec + ( zadnje_ucitano - *brojilo ) % TICKSPERSEC;
    if ( *usek >= TICKSPERSEC ) {
        *usek = *usec - TICKSPERSEC;
        *sek = *sek + 1;
    }
    return 0;
}

int postavi_alarm ( int sek, int usek, void (*funkcija)() )
{
    // provjere preskočene:
    // ( sek >= 0 && usek >= 0 && usek < 1000000 && funkcija != NULL )

    //relativan alarm: za {sek,usek} ga aktiviraj
    alarm_sec = sek;
    alarm_usec = usek;

    alarm = funkcija;

    prekid_brojila ();
}

void prekid_brojila ()
{
    //ažuriraj sat
    sat_sec += ( zadnje_ucitano - *brojilo ) / TICKSPERSEC;
    usec += ( zadnje_ucitano - *brojilo ) % TICKSPERSEC;
    if ( usec >= TICKSPERSEC ) {
        usec -= TICKSPERSEC;
        sec++;
    }

    zadnje_ucitano = MAXCNT;
    *brojilo = zadnje_ucitano;

    if ( alarm != NULL ) {
        if ( alarm_sec + alarm_usec == 0 ) {
            void (*tmp)() = alarm;
            alarm = NULL;
            tmp ();
        }
        else {
            if ( alarm_sec * TICKSPERSEC + alarm_usec < MAXCNT )
                zadnje_ucitano = alarm_usec + alarm_sec * TICKSPERSEC;
```

```
            //else zadnje_ucitano = MAXCNT; -- već prije postavljeno

            //koliko još ostane za idući put?
            alarm_sec -= zadnje_ucitano / TICKSPERSEC;
            alarm_usec -= zadnje_ucitano % TICKSPERSEC;
            if ( alarm_usec < 0 ) {
                alarm_sec--;
                alarm_usec += TICKSPERSEC;
                if ( alarm_sec < 0 ) {
                    //greška u nepreciznosti; idući prekid je alarm
                    alarm_sec = 0;
                    alarm_usec = 0;
                }
            }
        }
    }
}
```

---

22. Neku napravu (`uredjaj_X`) treba dodati u sustav preko `device_t` sučelja:

```
struct _device_t_;
typedef struct _device_t_ device_t;

struct _device_t_
{
   /* device interface */
   int (*init) ( uint flags, void *params, device_t *dev );
   int (*destroy) ( uint flags, void *params, device_t *dev );
   int (*send) ( void *data, size_t size, uint flags, device_t *dev );
   int (*recv) ( void *data, size_t size, uint flags, device_t *dev );
};
```

Neka su operacije nad napravom ostvarene funkcijama (te funkcije postoje!):

```
int uredjaj_X_posalji ( void *sto, size_t koliko );
int uredjaj_X_primi ( void *kamo, size_t koliko );
```

Definirati sučelje (varijablu `uredjaj_x` tipa `device_t`) za tu napravu te ju inicijalizirati u funkciji `inicijaliziraj_sucelje()`. Sve dodatne potrebne funkcije za to sučelje također napisati. Pretpostaviti da nije potrebna nikakva inicijalizacija naprave.

---

23. U nekom sustavu sučelje za rad s napravama jest:

```
struct naprava_t
{
    int (*init) ( struct naprava_t *n );
    int (*send) ( struct naprava_t *n, void *data, size_t size );
    int (*recv) ( struct naprava_t *n, void *data, size_t size );
    void *param;
};
```

Napisati upravljački program za napravu X korištenjem gornjeg sučelja. Pretpostaviti da je naprava dostupna na adresama: S (za slanje), R (za čitanje) i C (za statusni registar). Čitanjem podatka na adresi C dobiva se status naprave. Ako je prvi bit pročitanog broja postavljen onda se s adrese R može pročitati idući podatak (ima ga). Ako je drugi bit postavljen može se napravi poslati novi podatak (ona će ga moći prihvatiti). Radi ubrzanja rada za ulaz i izlaz dodati međuspremnike kapaciteta 4096 B (rezervirati ih s `malloc()`) i koristiti ih za pohranjivanje novih podataka iz naprave, odnosno za privremenu pohranu kada se podaci ne mogu proslijediti prema napravi. Operacije `send` i `recv` trebaju koristiti te međuspremnike (u

prethodno opisanim situacijama).

Rješenje:

```
#define MS          4096
#define int8        unsigned char

struct ms
{
    int8 bi[MS], bo[MS];
    int bi_f, bi_l, bi_sz, bo_f, bo_l, bo_sz;
};

static void x_interrupt_handler ( struct naprava_t *n );

static int x_init ( struct naprava_t *n )
{
    n->param = malloc (sizeof (struct ms) );
    memset ( n->param, 0, sizeof (struct ms) );
    registriraj_prekid ( X, x_interrupt_handler, n );
}

void x_interrupt_handler ( struct naprava_t *n )
{
    struct ms *ms = n->param;
    int8 *s = (int8 *) S, *r = (int8 *) R, *c = (int8 *) C;

    while ( ms->bi_sz < MS && ( (*c) & 1 ) ) {
        ms->bi[ms->bi_l] = *r;
        ms->bi_sz++;
        ms->bi_l = ( ms->bi_l + 1 ) % MS;
    }
    while ( ms->bo_sz > 0 && ( (*c) & 2 ) ) {
        *s = ms->bo[ms->bo_f];
        ms->bo_sz--;
        ms->bo_f = ( ms->bi_f + 1 ) % MS;
    }
}

int x_send ( struct naprava_t *n, void *data, size_t size )
{
    struct ms *ms = n->param;
    int8 *d = data, sz = size;
    int8 *s = (int8 *) S, *c = (int8 *) C;

    //prvo probaj poslat izravno na napravu
    while ( ms->bo_sz == 0 && ( (*c) & 2 ) && sz > 0 ) {
        *s = *d;
        d++;
        sz--;
    }
    //ostatak u ms
    for ( ; sz > 0 && ms->bo_sz < MS; ) {
        ms->bo[ms->bo_l] = *d;
        d++;
        sz--;
        ms->bo_sz++;
        ms->bo_l = ( ms->bo_l + 1 ) % MS;
    }

    if ( sz > 0 )
        return size - sz; //toliko je ukupno poslano i stavljeno u ms
}

int x_recv ( struct naprava_t *n, void *data, size_t size )
{
```

```
    struct ms *ms = n->param;
    int8 *d = data, sz = size;
    int8 *r = (int8 *) R, *c = (int8 *) C;

    //prvo čitaj iz ms
    for ( ; ms->bi_sz > 0 && sz > 0; ) {
        *d = ms->bi[ms->bo_f];
        d++;
        sz--;
        ms->bi_sz--;
        ms->bi_f = ( ms->bi_f + 1 ) % MS;
    }
    //sada probaj čitat izravno s naprave
    while ( ( (*c) & 1 ) && sz > 0 ) {
        *d = *r;
        d++;
        sz--;
    }

    if ( sz > 0 )
        return size - sz; //toliko je pročitano
}

/* sučelje */
struct naprava_t x = { .init = x_init, .send = x_send, .recv = x_recv };
```

---

24. Gdje se sve nalaze kopije jedne datoteke koja je u sustavu koji koristi *git*? Pretpostaviti da se radi o jednom projektu – jednom git repozitoriju na poslužitelju te jednog korisnika koji ga koristi (dohvatio ga je s `git clone ...`).

Rješenje:

1. u repozitoriju na poslužitelju
2. u lokalnom repozitoriju (.git direktoriju)
3. u lokalnoj kopiji ("radna inačica")

---

25. Radi provjere ispravnog rada u kod se ugrađuju dodatne provjere. Neke od njih se izvode samo u ispitnom pokretanju ('DEBUG' načinu), a neke uvijek. Pokazati na primjeru potrebu korištenja oba načina provjera.

Rješenje:

Ispitno pokretanje izvodi se pri razvoju sustava, dok je vjerojatnost postojanja grešaka veća. Stoga se u tim pokretanjima koriste dodatne direktive pri prevođenju (DEBUG) čime se uključuju dodatne provjere, pogotovo na početku funkcija gdje se dodatno provjeravaju poslani parametri. Primjeri takvih ispitivanja su:

```
int neka_funkcija ( tip1 p1, tip2 p2, ... )
{
    ASSERT ( "provjera p1" ); //ili assert
    ...
}
```

U fazi pravog rada sustava i dalje se mogu provjeravati neki parametri, pogotovo oni koji dolaze iz programa. Takve provjere moraju biti ostvarene običnim kodom (ne makroima ASSERT i sl.). Dodatne provjere tijekom pravog rada uglavnom se svode na probleme nedostatka sredstava i grešaka u radu (koje najčešće nisu rezultat krivog programa već ulaza i sl.). Primjerice, svaki bi zahtjev za stvaranje nekog objekta (npr. dretve) ili zahtjev za dijelom spremnika (malloc)

trebalo provjeriti.

```
int neka_druga_funkcija ( tip1 p1, tip2 p2, ... )
{
    if ( "provjera p1" ) {
        "ili vrati grešku ili prekini dretvu ili ...";
    }
    ...
    x = malloc (...);
    if ( x == NULL ) {
        "prijavi grešku u neki dnevnik ili korisniku preko zaslona, ..."
        "ili vrati grešku ili prekini dretvu ili ...";
    }
    ...
}
```

---

26. Navesti primjere gdje jedan program (dretva) zbog greške može narušiti cijeli sustav. Kojim mehanizmima se navedeni problemi mogu lokalizirati (da greške ne utječu na ostatak sustava)?

Rješenje:

Zbog mijenjanja podataka drugih procesa ili jezgre OS-a. Rješenje: koristiti upravljanje spremnikom koje će onemogućiti procesu da piše van ograđenog prostora.

Zbog instrukcija koje upravljaju nekim dijelovima sustava. Primjerice, program može zabraniti prekide i time onemogućiti prihvat prekida svih naprava (a time i sata koji bi tu dretvu maknuo s procesora). Rješenje: procese izvoditi u korisničkom načinu rada u kojem ne mogu pokretati takve instrukcije.

---

27. Neki sklop detektira otkucaje sata te preko prikladnog sučelja zapisuje broj 1 na adresu `BEAT`. Korištenjem tog podatka ostvariti sustav koji će na zaslonu uređaja prikazivati:

1. ukupan broj otkucaja (`b`)
2. trenutnu frekvenciju otkucaja (broj otkucaja u minuti `bpm`)
3. procjenjenu potrošnju kalorija po minuti (računati preko `cpm=fun1(bpm)`, `fun1` postoji)
4. procjenjeni ukupan broj potrošenih kalorija (`cals=fun2(b)`, `fun2` postoji).

Sustav posjeduje 16-bitovno brojilo koje odbrojava frekvencijom FREQ. Brojilo nije moguće promijeniti (resetirati) niti ono izaziva prekide, već nastavlja s nulom nakon što dosegne najveću vrijednost.

Pretpostaviti da se ispis vrijednosti na zaslon zbiva jednostavnim upisom odgovarajućih vrijednosti na zasebne lokacije u spremniku (adrese: `B`, `BPM`, `CPM`, `CALS`). Broj otkucaja po minuti izračunavati na osnovu zadnja četiri otkucaja: $bpm = 60 \cdot 3/(t_4 - t_1)$. Upravljanje ostvariti upravljačkom petljom (beskonačnom petljom).

Rješenje:

```
//struktura podataka:
#define MAXCNT  (1<<16)
int b, bpm;
int *pb = B, *pbpm = BPM, *pcpm = CPM, *pcals = CALS;
int *beat = BEAT, *brojilo = BROJILO;
int t[4] = {0,0,0,0}, T;

//upravljačka petlja
void upravljanje ()
{
    while (1) {
```

```
        if ( *beat ) {
            *beat = 0;
            b++;

            t[0] = t[1]; t[1] = t[2]; t[2] = t[3];
            t[3] = *brojilo;

            T = 0;
            if ( t[1] > t[0] )
                T += t[1] - t[0];
            else
                T += t[1] + MAXCNT - t[0];

            if ( t[2] > t[1] )
                T += t[2] - t[1];
            else
                T += t[2] + MAXCNT - t[1];

            if ( t[3] > t[2] )
                T += t[3] - t[2];
            else
                T += t[3] + MAXCNT - t[1];

            bpm = 60 * 3 * FREQ / T;

            *pb = b;
            *pbpm = bpm;
            *pcpm = fun1(bpm);
            *pcals = fun2(b);
        }
    }
}
```

---

28. Neko ugrađeno računalo ima brojilo koje odbrojava od zadane vrijednosti do nule s frekvencijom od 1 MHz. Najveća vrijednost koja stane u brojilo jest 100000. Kada brojilo dođe do nule izaziva prekid broj 50, u brojilo automatski učitava 100000 te nastavlja s odbrojavanjem. Prekidni podsustav nudi sučelje `void register_interrupt(irq_num, handler)`. Neka se sadržaj brojila može dohvatiti sa `int cnt_get()` a postaviti s `void cnt_set(int value)`. Ostvariti sustav praćenja vremena, tj. sučelje `void dohvati_trenutno_vrijeme(int *sec, int *usec)` koje vraća trenutno vrijeme s preciznošću od mikrosekunde (uz pretpostavku brzog procesora, tj. zanemarenja trajanja izvođenja funkcija za dohvat vremena) te sustav upravljanja alarmima sa sučeljem `void postavi_alarm(void (*akcija)(), int sec)` koje postavlja alarm koji treba aktivirati (pozvati zadanu funkciju) za zadani broj sekundi. Definirati sve potrebne strukture podataka i funkcije. Pretpostaviti da postoje operacije za rad s listama te za dinamičko dohvaćanje spremnika (za potrebe opisnika svih alarma).

---

29. U nekom računalnom sustavu postoji 64-bitovno brojilo koje odbrojava frekvencijom od 1 GHz. Korištenjem tog brojila ostvariti:

a) praćenje dodijeljenog procesorskog vremena pojedinoj dretvi

b) raspoređivanje podjelom vremena.

Pretpostaviti da svaki poziv jezgrine funkcije započinje operacijom `deaktiviraj_dretvu` iz koje se poziva funkcija `ažuriraj_vremena` koju treba ostvariti (za a dio).

Nadalje, pretpostaviti da u sustavu postoji i drugi satni mehanizam koji periodički izaziva prekide (dovoljno velikom, ali nepoznatom i nestalnom frekvencijom). Iz tih se funkcija poziva `raspoređivanje_podjelom_vremena` koju treba ostvariti (za b dio). Neka sve dre-

tve trebaju dobiti kvant vremena `T`. Za samo raspoređivanje ne koristiti dodatne alarme već samo poziv `raspoređivanje_podjelom_vremena` koji je ugrađen u sve potrebne funkcije. Za praćenje dobivenog vremena po dretvi koristiti podatke prikupljene u a) dijelu zadatka. Zbog poziva iz drugih funkcija poneka će dretve dobiti i više vremena od T, ali manje ne smije. Za upravljanje dretvama koristiti pozive: `stavi_u_pripravne(dretva)`, `prva=uzmi_prvu_pripravnu()`, `aktivna=dohvati_aktivnu()` te `postavi_aktivnu(dretva)`.

Proširiti opisnike dretve po potrebi. Korištenje vremena pojednostaviti (npr. koristiti vrijeme u jedinicama nanosekunde u 64-bitovnim varijablama).

Rješenje: a)

1) uz pretpostavku da jezgrine funkcije traju kratko (zanemarivo).

```
#define BROJILO neka_adresa //neka brojilo odbrojava prema većim vrijednostima
#define FREQ    1000000000
#define MAX 0xffffffffffffffff  //2^64-1
long zadnje_očitanje_brojila = 0;
long *brojilo = BROJILO;

ažuriraj_vremena () //po deaktivaciji dretve
{
    aktivna = dohvati_aktivnu();
    if ( *brojilo > zadnje_očitanje_brojila )
        t = *brojilo - zadnje_očitanje_brojila;
    else
        t = MAX - zadnje_očitanje_brojila + 1 + *brojilo;

    zadnje_očitanje_brojila = *brojilo;
    aktivna->dobiveno_vremena += t;
}
```

2) uz pretpostavku da jezgrine funkcije traju duže (nezanemarivo) (informativno)

```
#define BROJILO neka_adresa //neka brojilo odbrojava prema gore
#define FREQ    1000000000
#define MAX 0xffffffffffffffff  //2^64-1
long *brojilo = BROJILO;

ažuriraj_vremena ()
{
    aktivna = dohvati_aktivnu();
    if ( *brojilo > aktivna->zadnje_očitanje_brojila )
        t = *brojilo - aktivna->zadnje_očitanje_brojila;
    else
        t = MAX - aktivna->zadnje_očitanje_brojila + 1 + *brojilo;

    aktivna->dobiveno_vremena += T;
}
prije_povratka_u_dretvu (aktivna)
{
    aktivna->zadnje_očitanje_brojila = *brojilo;
}
```

Rješenje: b)

```
raspoređivanje_podjelom_vremena ()
{
    ažuriraj_vremena ();
    aktivna = dohvati_aktivnu();
    t = aktivna->dobiveno_vremena - aktivna->dobiveno_prije;
    if ( t >= T ) {
        stavi_u_pripravne ( aktivna );
        aktivna = uzmi_prvu_pripravnu ();
```

```
        postavi_aktivnu ( aktivna );
        aktivna->dobiveno_prije = aktivna->dobiveno_vremena;
    }
}
```

---

30. Prikazati postupak ostvarenja jezgrine funkcije `int dohvati_vrijeme ( time_t * t )` koja dohvaća trenutno vrijeme sustava i pohranjuje ga na adresu `t`. Pretpostaviti da se programi izvode u logičkom adresnom prostoru (adresni prostor procesa kreće od adrese 0) dok se jezgrine funkcije izvode s fizičkim adresama. Prikazati i funkciju i pomoćne funkcije i strukture podataka. Odabir načina prijenosa parametara u jezgrinu funkciju je proizvoljan (npr. stog ili zasebne varijable u dretvi). Interna jezgrina funkcija za dohvat vremena `int j_dohvati_vrijeme ( time_t *t )` radi s fizičkim adresama.

---

31. U sustavu koji koristi sklopovsku potporu za dinamičko upravljanje spremnikom, programi koriste logičke adrese dok u se u jezgrinim funkcijama koriste fizičke (apsolutne). Ostvariti jezgrinu funkciju `j_najveća` koja za zadana imena datoteka vraća ime i veličinu najveće datoteke (od zadanih imena). Neka je funkcija koja se poziva iz programa:

```
int najveća ( char **imena, size_t *veličina, char **najveća )
{
    izazovi_programski_prekid;
}
```

Iz obrade programskog prekida poziva se jezgrina funkcija: `j_najveća(void *parametri )` gdje je jedini parametar adresa (fizička) stoga pozivajuće dretve (adresa gdje se nalazi prvi parametar `imena`). Izvođenje jezgrine funkcije obavlja se korištenjem fizičkog načina adresiranja (pretvaranje adresa je isključeno). Uz pretpostavku da postoji pomoćna funkcija `vrati_veličinu(ime)` koja vraća veličinu zadane datoteke, ostvariti jezgrinu funkciju `j_najveća`. Zanemariti povratnu vrijednost funkcije (ono što `najveća` vraća kao povratnu vrijednost, ne preko parametara). Adresu početka spremničkog prostora trenutnog procesa može se dohvatiti s `početna_adresa_procesa(NULL)`.

Rješenje:

```
int j_najveća ( void *parametri )
{
    char **imena;
    size_t *veličina;
    char **najveća;

    imena = *((char ***) parametri); parametri += sizeof (char **);
    veličina = *((size_t **) parametri); parametri += sizeof (size_t *);
    najveća = *((char ***) parametri);

    //adrese su u logičkom obliku, pretvori ih u apsolutni
    početna = početna_adresa_procesa ( NULL );
    imena += početna;
    veličina += početna;
    najveća += početna;

    max = -1;
    imax = 0;
    for ( i = 0; imena[i] != NULL; i++ ) {
        ime = imena[i] + početna; //pretvori u fiz. adresu
        vel = vrati_veličinu ( ime );
        if ( vel > max ) {
            imax = i;
            max = vel;
        }
```

```
    }
    *veličina = max;
    *najveća = imena[imax]; //ovo već je u logičkim adresama
}
```

Literatura

Izvorni kôdovi na koje se ova skripta odnosi:

[Benu] *Benu* – izvorni kôdovi, https://github.com/ljelenkovic/Benu

Knjige i skripte:

[Budin, 2010] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenković, *Operacijski sustavi*, Element, 2010.

[Silberschatz, 2002] Abraham Silberschatz, Greg Gagne, Peter Baer Galvin, *Operating System Concepts, 6th edition*, Wiley, 2002.

[Tanenbaum, 1987] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.

[Labrosse, 2002] J.J. Labrosse, *MicroC/OS-II: The Real Time Kernel, 2nd edition*, CMP Books, San Francisco, 2002.

[Rajkumar, 1991] R. Rajkumar, *Synchronization in real-time systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.

[Wolf, 2001] W. Wolf, *Computers as Components – Principles of Embedded Computing System Design*, Academic Press, 2001.

[Bar, 1999] M. Barr, *Programming Embedded Systems in C and C++*, O'Reilly & Associates, 1999.

[Nutt, 2000] G. Nutt, *Operating Systems: A Modern Perspective*, Addison-Wesley, Reading, 2000.

[SRSV, 2012] Leonardo Jelenković, *Sustavi za rad u stvarnom vremenu*, skripta za predavanje, 2013. http://www.fer.unizg.hr/predmet/szrusv

Alati, priručnici za korištenje alata, tehnički opisi sklopovlja:

[GCC] *GCC, the GNU Compiler Collection*, http://www.gnu.org/software/gcc.

[Binutils] *GNU Binutils*, http://www.gnu.org/software/binutils.

[Binutils2] *GNU Binutils – Source Code Reference*, https://sourceware.org/binutils/docs/ld/Source-Code-Reference.html.

[Make] *GNU Make*, http://www.gnu.org/software/make.

[ARM-EABI] *Sourcery CodeBench Lite Edition (for ARM EABI)*, http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/.

[GRUB] *GNU GRUB*, http://www.gnu.org/software/grub.

[GDB] *GDB: The GNU Project Debugger*, http://www.gnu.org/software/gdb.

[QEMU] *QEMU - open source processor emulator*, http://wiki.qemu-project.org.

[Git] Git – source control management, https://git-scm.com.

[VMware] *VMware Player*, http://www.vmware.com/products/player.

[VirtualBox] *VirtualBox*, https://www.virtualbox.org.

[Intel, 2009] *Intel® 64 and IA-32 Architectures Software Developer's Manual – Vol: 1, 2A, 2B, 3A, 3B*, 2009. http://www.intel.com/products/processor/manuals.

[ARM926EJ-S] *Versatile Application Baseboard for ARM926EJ-S User Guide*, 2011. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0225d/index.html.

[vexpress-a9] *Versatile Express*, 2015. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0448i/index.html.

[gcc-interrupt] *GCC – x86 Function Attributes*, 2019. https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html.

[POSIX] IEEE and Open Group, *The Open Group Base Specifications Issue 7*, http://pubs.opengroup.org/onlinepubs/9699919799/.

[ELF] *Executable and Linkable Format* (stari naziv: *Extensible Linking Format*), http://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

Upute, forumi i slično, vezani uz problem izgradnje operacijskih sustava

[OSDev.org] *OS Development*, http://wiki.osdev.org

[Bona Fide] *Bona Fide OS Development*, http://www.osdever.net

[BrokenThorn] *Operating System Development Series*, http://www.brokenthorn.com/Resources/OSDevIndex.html

Internet stranice s opisom sklopovlja/algoritama:

[Intel 8259] http://en.wikipedia.org/wiki/Intel_8259, http://wiki.osdev.org/PIC, http://www.thesatya.com/8259.html, http://stanislavs.org/helppc/8259.html.

[Intel 8253] http://en.wikipedia.org/wiki/Intel_8253, http://www.sharpmz.org/mz-700/8253ovview.htm, http://stanislavs.org/helppc/8253.html.

[osdev.org/CS] http://wiki.osdev.org/Context_Switching.

[Tipkovnica] *Operating Systems Development – Keyboard*: http://www.brokenthorn.com/Resources/OSDev19.html,
PS2 Keyboard: http://wiki.osdev.org/PS2_Keyboard,
IBM PC keyboard information for software developers: http://www.osdever.net/documents/pdf/kbd.pdf,
Keyboard scancodes: http://www.win.tue.nl/~aeb/linux/kbd/scancodes.html,
IBM AT and XT Keyboards Operation: http://vgcontrols.com/Products/encoders/pdf/pckeybrd.pdf,
PC Keyboard Scan Codes: http://www.barcodeman.com/altek/mule/scandoc.php,
The AT-PS/2 Keyboard Interface, Adam Chapweske: http://www.tayloredge.com/reference/Interface/atkeyboard.pdf.

[Operator ##] https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html,
http://en.wikipedia.org/wiki/C_preprocessor#Token_concatenation,
https://msdn.microsoft.com/en-us/library/wy090hkc(v=vs.140).aspx.

[Multiboot] http://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[dlmalloc] Doug Lee, *A Memory Allocator*, http://g.oswego.edu/dl/html/malloc.html.

[TLSF] http://www.gii.upv.es/tlsf/.

[Murphy, 2000] N. Murphy, *How to Use Watchdog Timers Properly when Multitasking*, November 2000, https://barrgroup.com/Embedded-Systems/How-To/Advanced-Watchdog-Timer-Tips.

[Murphy, 2001] N. Murphy, M. Barr, *Introduction to Watchdog Timers*, October 2001, https://barrgroup.com/Embedded-Systems/How-To/Watchdog-Timer.

[restrict] Ključna riječ *restrict*, http://en.wikipedia.org/wiki/Restrict.

[ANSI code] Upravljački kodovi: *ANSI escape code*, http://en.wikipedia.org/wiki/ANSI_escape_code.

Operacijski sustavi za ugrađene sustave i povezani materijali:

[Jelenković, 2005] L. Jelenković, *Višedretveni ugrađeni sustavi zasnovani na monitorima*, doktorska disertacija, 2005.

[Budin, 1999] L. Budin, L. Jelenkovic, *Time-Constrained Programming in Windows NT Environment*, IEEE International Symposium on Industrial Electronics ISIE'99, Vol. 1, pp. 90-94, Bled, 1999.

[Engler, 1998] D. Engler, *The Exokernel Operating System Architecture*, PhD thesis, Massachusetts Institute of Technology, 1998.

[Jones, 1997] M. Jones, *What really happened on Mars?*, http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html.

[Laurich, 2004] Peter Laurich, *A comparison of hard real-time Linux alternatives*, 2004, http://www.linuxdevices.com/articles/AT3479098230.html.

[Schmidt, 2002] Douglas C. Schmidt, Mayur Deshpande, and Carlos O'Ryan, *Operating System Performance in Support of Real-time Middleware*, http://www.cs.wustl.edu/~schmidt/PDF/words-02.pdf.

[Williams, 2008] Clark Williams, *An Overview of Realtime Linux*, 2008, http://people.redhat.com/bche/presentations/realtime-linux-summit08.pdf.

[T-T debate] *The Tanenbaum-Torvalds Debate*, http://www.oreilly.com/catalog/opensources/book/appa.html.

[Linux] *Izvorni kod jezgre Linuxa*, https://www.kernel.org/.

[BusyBox] *BusyBox*, http://www.busybox.net.

[U-Boot] *Das U-Boot – the Universal Boot Loader*, http://www.denx.de/wiki/U-Boot.

[uClinux] *Embedded Linux/Microcontroller Project*, http://www.uclinux.org/.

[Buildroot] *Buildroot: making Embedded Linux easy*, https://buildroot.org/.

[Neutrino] *Neutrino Realtime Operating System*, http://www.qnx.com/products/rtos.

[VxWorks] *VxWorks real time system*, http://windriver.com/products/vxworks.

[Windows EC] *Windows Embedded Compact 7 (Formerly CE)*, http://www.microsoft.com/windowsembedded/en-us/evaluate/windows-embedded-compact-7.aspx.

[RTLinux] *RTLinux*, http://en.wikipedia.org/wiki/RTLinux.

[RT Linux] *Real-Time Linux*, https://wiki.linuxfoundation.org/realtime/start.

Poveznice na opise uobičajenih (POSIX) funkcija

Navedene poveznice mogu se naći tražilicom na [POSIX].

open
close
read
write
printf
pthread.h
pthread_create
pthread_exit
pthread_join
pthread_self
pthread_attr_init
pthread_attr_destroy
pthread_attr_setschedpolicy
pthread_setschedparam
pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_timedlock
pthread_mutex_unlock
pthread_mutexattr_init
pthread_mutexattr_destroy
pthread_cond_init
pthread_cond_destroy
pthread_cond_wait
pthread_cond_timedwait
pthread_cond_signal
pthread_cond_broadcast
pthread_condattr_init
pthread_condattr_destroy
semaphore.h
sem_init
sem_destroy
sem_post
sem_wait
sem_wait
sem_wait
mqueue.h
mq_open
mq_close
mq_send
mq_receive
time.h
clock_gettime
clock_settime
clock_nanosleep
nanosleep
signal.h
timer_create
timer_delete
timer_settime
timer_gettime
sigaction
sigqueue
sigwaitinfo
pthread_sigmask
posix_spawn